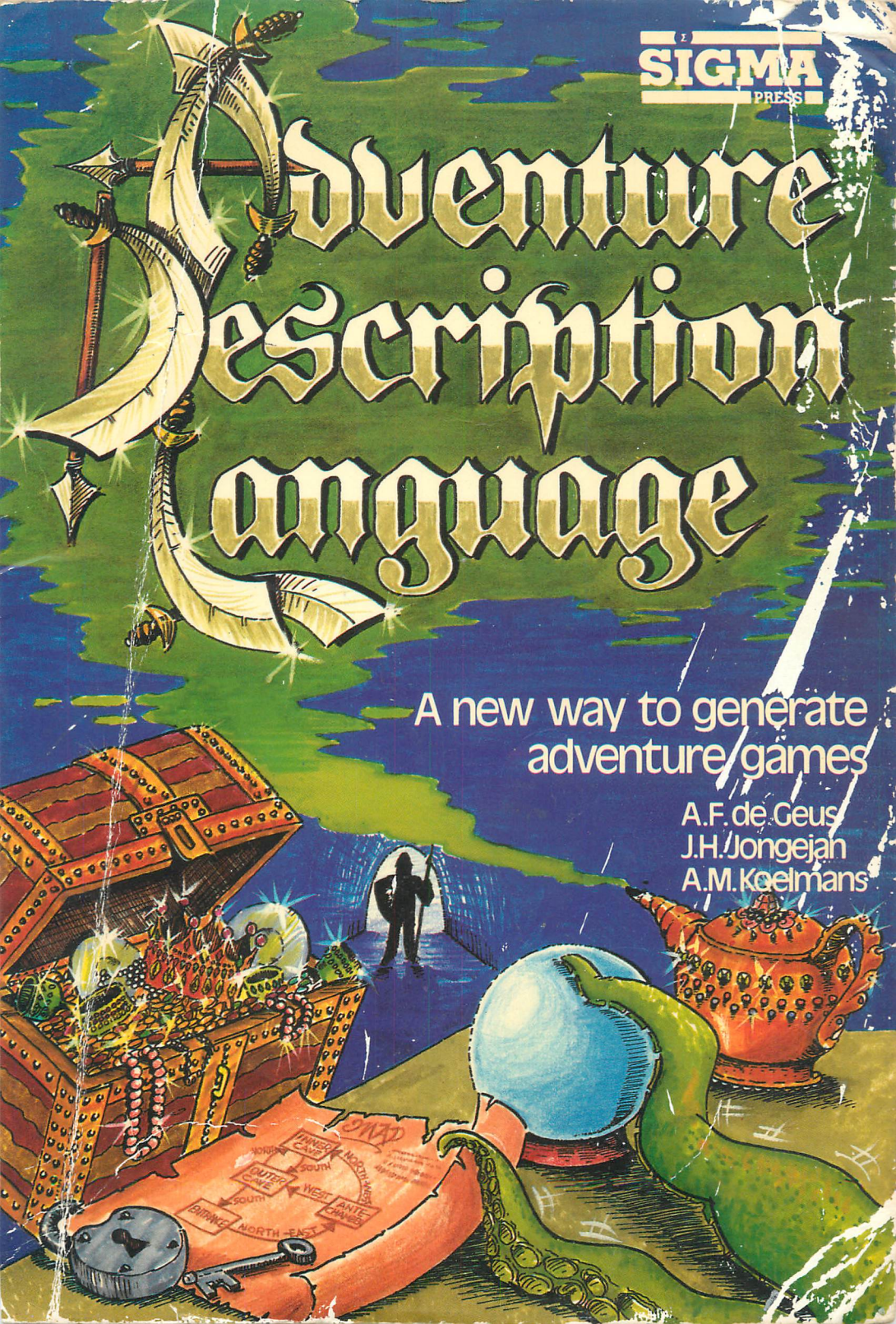
SIGMA PRESS

Adventure Description Language

A new way to generate adventure games

A.F. de Geus
J.H. Jongejan
A.M. Koelmans



Adventure Description Language

—a new way to generate adventure programs

A. F. de Geus, J. H. Jongejan
and A. M. Koelmans

Σ
SIGMA
PRESS

Copyright © A. F. de Geus, J. H. Jongejan and A. M. Koelmans

All Rights Reserved

No part of this book may be reproduced or transmitted by any means without the prior permission of the publisher. The only exceptions are for the purposes of review, or as provided for by the Copyright (Photocopying) Act or in order to enter the programs herein onto a computer for the sole use of the purchaser of this book.

ISBN 1 85058 011 1

Published by:
SIGMA PRESS
5 Alton Road
Wilmslow
Cheshire
U.K.
Printed in Malta by
Interprint Limited

Distributors:

U.K., Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester
West Sussex, England

Australia:
JOHN WILEY & SONS INC.
GPO Box 859, Brisbane
Queensland 40001
Australia

Preface

The subject of this book, an adventure generator, was developed by two of the authors during 1980 while graduate students at the University of Groningen, The Netherlands. It was written in the programming language C, under the UNIX™ timesharing system on a DEC PDP11 minicomputer. We found it so pleasant to use that we decided to make it available to a larger audience. Hence this book, which presents a version for the BBC Micro.

The software will only run on a machine with a disc unit, because several files need to be open simultaneously. It was developed on a BBC model B with OS 1.2 and Basic 2 ROM fitted. It has not been tried on a machine with Basic 1, so beware. You certainly will not be able to assemble the interpreter source code from Basic 1, because extensive use of EQU directives and the relocation facilities is made.

We have done our utmost to test the software as thoroughly as possible, but should bugs appear, our apologies. Please drop a note to the publisher if you should find any. If you do not want to type in the listings from the book, you can purchase the software separately.

A disk containing all of the software used in this book can be obtained directly from Sigma Press for £7.95 inclusive of VAT and postage. This is a reader convenience disk, no frills, no fancy packaging.

Groningen/Newcastle, January 1985.

Fred de Geus
Jan Jongejan
Albert Koelmans

UNIX is a trademark of Bell Laboratories.

Contents

CHAPTER I: Introduction 1

CHAPTER II: The Adventure Description Language 4
II.1 Introduction 4
II.2 An adventure game 5
II.3 The dictionary 7
II.4 The transition graph 8
II.5 The strings 10
II.6 A special purpose adventure language 11
II.6.1 Introduction 11
II.6.2 Why an adventure language? 11
II.6.3 Adventure language implementation 12
II.7 Verbs and Objects 15
II.7.1 Intransitive verbs 15
II.7.2 Transitive verbs 15
II.7.3 Conditional and random transitions 16
II.7.4 Objects 17
II.7.5 Other ingredients 18
II.8 The Adventure Description Language 20
II.8.1 Introduction
II.8.2 Main syntax constructions 20
II.8.3 The statement 21
II.8.4 The IF statement 22
II.8.5 Conditions 23
II.8.6 The CASE statement 23
II.8.7 Action procedures and condition functions 24
II.8.8 Examples 26
II.9 Adventure game implementation using ADL 29
II.9.1 The dictionary 29
II.9.2 The transition table 31
II.9.3 The strings 32
II.9.4 The syntax for verbs 33
II.9.5 The scanner 35
II.9.6 The interpreter 36
II.9.7 Conclusion 37

CHAPTER III: The Red Button: a small game written in ADL 39
III.1 Introduction 39
III.2 The basic idea 40
III.3 The map 41
III.4 The first puzzle 44

III.5 Extending the login procedure 45
III.6 The user identification codes 47
III.7 The coffee machine 50
III.8 Hiding the coin 53
III.9 Intermezzo: help commands 56
III.10 Extending the "run" command 58
III.11 The magnetic media 60
III.12 Loading the shutdown program 62
III.13 The door to the secret bunker 64
III.14 Drinking coffee 66
III.15 Completing the game 68
III.16 Random transitions and events 71
III.17 Recursive definitions 76
III.18 Putting it all together 78

CHAPTER IV: The theory behind the Adventure Description Language 79
IV.1 What is a grammar? 79
IV.1.1 Languages, sentences and words 79
IV.1.2 Construction of sentences 79
IV.1.3 Grammars 80
IV.1.4 An example of a language with a grammar 80
IV.1.5 An example of a grammar with its language 81
IV.1.6 Abstract definition of a grammar 82
IV.1.7 Generation and parsing of sentences 83
IV.1.8 A hierarchy of families of grammars 85
IV.2 Representation with BNF 87
IV.2.1 History of BNF 87
IV.2.2 Language and metalanguage symbols 87
IV.2.3 BNF Rules 87
IV.2.4 An example grammar in BNF 88
IV.3 What is an automaton? 90
IV.3.1 Abstracting away from the black box machine 90
IV.3.2 A simple example of an automaton 91
IV.3.3 Abstract definition of an automaton 91
IV.3.4 Representation of an automaton 92
IV.4 Automaton and grammars 93
IV.4.1 Finite State Generator 93
IV.4.2 Finite State Recogniser 96
IV.4.3 Nondeterministic machines 97
IV.4.4 Converting an automaton into BASIC 100
IV.5 The ADL revisited 103
IV.5.1 ADL is a context free language 103
IV.5.2 Checking against errors 104
IV.5.3 The ADL scanner and interpreter 105
IV.6 Some Theory about graphs 106
IV.6.1 Introduction 106
IV.6.2 Definition of a graph 106
IV.6.3 Walking around 107

IV.6.3.1 The Warsall algorithm 108
IV.6.3.2 Hamiltonian paths 109
IV.6.4 Graphs divided into subclasses 110
IV.6.4.1 Completeness 110
IV.6.4.2 Connectedness 111
IV.6.4.3 Cyclicity 111
IV.6.4.4 Trees 112
IV.6.4.5 Planar graphs 113
IV.7 Text compression 114
IV.7.1 Introduction 114
IV.7.2 Huffman Code 114
IV.7.2.1 How does it work? 115
IV.7.2.2 Building the tree 118
IV.7.2.3 Encoding and decoding 118
IV.8 Hashing 119
IV.9 Open hashing 120
IV.10 Closed hashing 121

Appendix A: The syntax of ADL 123
Appendix B: The complete red button game listing 125
Appendix C: The game of long forgotten Arabia 135
Appendix D: Error messages 153
Appendix E: The screen editor 156
Appendix F: The scanner listing 178
Appendix G: The interpreter listing 203

CHAPTER 1
Introduction

"You are on the edge of a breath-taking view. Far below you is an active volcano, from which great gouts of molten lava come surging out, cascading back down into the depths. The glowing rock fills the farthest reaches of the cavern with a blood-red glare, giving everything an eerie, macabre appearance. The air is filled with flickering sparks of ash and a heavy smell of brimstone. The walls are hot to the touch, and the thundering of the volcano drowns out all other sounds. Embedded in the jagged roof far overhead are myriad twisted formations composed of pure white alabaster, which scatter the murky light into sinister apparitions on the walls. To one side is a deep gorge, filled with a bizarre chaos of tortured rock which seems to have been crafted by the devil himself. An immense river of fire crashes out from the depths of the volcano, burns its way through the gorge, and plummets into a bottomless pit far off to your left. To the right, an immmense geyser of blistering steam erupts continuously from a barren island in the center of a sulfurous lake, which bubbles ominously. The far right wall is aflame with an incandescence of its own, which lends an additional infernal splendor to the already hellish scene. A dark, foreboding passage exits to the south"

The above text is a location description which appeared in the legendary first adventure game. This adventure was written, way back in 1978, by two bored computer programmers, Willie Crowther and Don Woods. The object of the game was to explore a cave and collect all the treasures hidden inside it while fighting off dwarves, trolls, a pirate and various nasty animals, and finding your way through a maze or two. The game was a phenomenal success; versions surfaced all over the world, and many an hour of working time was lost because people who started to play were simply unable to stop! The original game was written in FORTRAN, on a DEC mainframe computer, and occupied a massive amount of memory. You can imagine the reactions when another computer programmer, Scott Adams, told his friends he was going to try to implement the game on his TRS-80 micro, with its tiny 16K memory! Nobody believed Adams would pull it off, but he did, and the rest is history. No doubt adventures are now among the most popular micro-computer games. Adams went on to write such classic adventures as "Pirate Adventure"and "Mission Impossible".

There are actually good reasons why people should be sceptical of the feasibility of implementing an adventure game of such a size on a micro with a limited memory. Most micros run BASIC as their resident programming language; on virtually all the machines it is an *interpreted* language. This

means that a BASIC program itself is not converted into machine code, but that somewhere within the computer (usually in ROM) there is a machine code program called the interpreter that executes the BASIC program. Thus the BASIC program itself needs to be resident in memory all the time, severely limiting the amount left for location descriptions (''you are in a maze of twisting little passages, all alike''), messages (''you can't go that way'') and object descriptions (''there's a rusty black rod lying here''). Thus adventures written in BASIC tend to be small and therefore perhaps not very interesting. It would certainly seem impossible to implement the original adventure on the BBC Micro: the Beeb has 32K of RAM, while the adventure program takes about 37K just for its strings!

All this may discourage you from trying to write your own adventures. You may be disinclined to do so anyway; most micro owners have never had any form of training in computer programming, and writing a relatively large program is beyond their capability. For them, attempting to write a large adventure game would simply result in an endless, frustating debugging session. This is not an expression of arrogance on our part; programming is a craft, and like all other crafts it takes a lot of exercise and study to learn it properly.

So, how then do the professional adventure writers manage to cram such a large amount of text into the Beeb's memory? What if you want to write and play your own adventures? Just put up with short, boring games and prolonged, even more boring debugging sessions? Do you have to go on spending money on yet another cassette?

The answer to the last two questions is no, and this book aims to explain how. We propose the following solution to the problem of designing and playing large adventures on a micro with a small memory:

- first, define a special purpose *adventure description language;*
- secondly, write a *compiler* for this language, i.e. write a program that will take the adventure ''program''and transform it into some internal form;
- finally, write an *interpreter* that will ''execute''the adventure program.

The purpose of the adventure language is to write your adventure in an easily readable and understandable form; there is no longer any need to worry about BASIC; instead you can put all your effort into the adventure itself.

The function of the compiler (hereafter to be called the *scanner*) is to squeeze the adventure into the smallest possible amount of memory. This applies in particular to the strings; by using a non-standard character code called the *Huffman Code,* string lengths can be reduced to about 50% of their original size. This makes it possible to reduce string files of 40K to about 20K, which of course fits easily into the Beeb's memory, with room to spare.
The scanner reads its input directly from disc, so it doesn't have any space problems.

The interpreter is a small program that reads your commands from the keyboard, determines which location transitions these commands cause, and then prints the appropriate message(s).

You will need some sort of text editor or word processor to build your adventure programs and string files. If you don't happen to have one, do not despair, but refer to Appendix E where a listing of a simple but powerful screen editor is shown.

We, the authors, are not quite satisfied with just presenting the listings of the scanner and the interpreter in this book. You might be tempted to think that it was written in an ad-hoc way, just for the purpose of playing adventures. In fact, these programs are based on proven methods of programming language design, developed by the computer science community over the past three decades. In the past, these methods have been applied to lots of programming languages, and no doubt will continue to be applied in the future. Just presenting you with a large program will obscure the fact that it is composed of a number of small steps, each of which can be more or less easily understood. So we decided to present the background theory to you as well, and show how it can be used to write a large program that does something really complicated. If you can grasp some of the theory - you may even be able to adapt the scanner and interpreter to suit your own needs!

To summarise, this book presents both an introduction to the formal theory of programming languages, and an interesting application of this theory, an adventure programming (and playing) system.
Chapter II presents the adventure language and the internal representation into which it is converted.
Chapter III defines a simple adventure game and shows how to program this game in the language defined in chapter II.
Chapter IV presents some formal theory of programming languages.
In the remaining part of the book you will find listings of the scanner and interpreter, and the listing of the screen editor.

The result of all this is a truly *professional* adventure system. We hope you will not be put off by the apparent complexity and size of the whole system; we are quite sure it is well worth the effort!

Happy adventures!

CHAPTER II

The Adventure description language

II.1. Introduction

Writing adventure games has two main aspects. The first one is the creative aspect: the construction of the adventure, its puzzles, how to lead and mislead a player, and how to speculate on the players intelligence. The second aspect is the implementation: how to put this particular adventure on your computer. Although the best way to create a game is to complete the first aspect before starting the second one, it is often difficult to elaborate a game in full detail. During the implementation new ideas arise and details are filled in. And when a game approaches its completion often several extensions have to be made.

The creative aspect is certainly the most interesting part, but the implementation can give you a lot of fun as well. The equipment and programming tools used, however, often impose considerable restrictions. It can become very tedious when you are not really free to make the game you would like, but can use only a subset of your original ideas.

With the growing role of small computers in our homes, the number of people that write their own adventure games has also rapidly increased. Although at first these could only be run on mainframe computers new techniques and increasing power of micro computers now allow the implementation of very large games on computers with 64 kilo bytes of memory or less.

Creating your own game is however not yet open to a large public, because the programming tools available on most microcomputers are still inadequate. In this chapter a new tool will be described to facilitate the writing of adventure games: an adventure description language.

But first a survey will be given of the main ingredients of an adventure, and the main difficulties encountered during implementation.

II.2. An adventure game

If we take a closer look at adventure games, we see that nearly all have things in common. We shall first give an intuitive description.

An adventure game plays in a number of locations, with the player starting in one of them. By giving commands to the computer, the player can move from one location to another. At certain locations objects can be found; some of these are aids to the player, others are treasures, and again others are part of puzzles that need to be solved.

To facilitate our description we will refer to persons and animals in a game as objects as well. Objects can be manipulated by giving special commands to the computer. The goal of an adventure game is to solve all of the puzzles; this can be the gathering of all treasures or the defeat of some evil ruler.

We will now give a more or less precise description of an adventure game which will provide us with an easier approach towards a computer implementation. Moving from one location to another will be called a *transition.*

All of the locations of a game together with all possible transitions between them will be called the *transition graph.* A command that causes a transition will be called a *move verb.* All the words known to the computer form the *dictionary* of the game.

Besides the move verbs and the objects two other categories will be used in our description of the dictionary:

- the transitive verbs
- the intransitive verbs.

Verbs of the first group can be applied to an object, for example "kill" (someone), "take" (object) etc.; verbs of the second group cannot be applied to an object, for example "look", "help", inventory etc..

In our model two more things are needed:

1) The messages which are sent to the player by the game. Mes sages can be divided in three groups: the descriptions of locations, the descriptions of objects and the remaining gen eral messages. These messages will be called *strings.*
2) The meaning of the different verbs (or commands) of the player that do not cause a transition. In most adventures this meaning is programmed in the available computer language, in general BASIC or machine code. The central theme of this book is to describe the meaning of verbs by a special adventure description language and we will describe the

advantages this has for an adventure writer. The description of verbs in this language will be called the *syntax* of the game.

This completes our model of an adventure; the elements are:

- the transition graph, which describes the transitions from one location to the other,
- the dictionary, which describes which inputs typed by the player are legal,
- the strings; these are output from the game to the player,
- the syntax, which describes the meaning of input typed by the player which does not cause a transition.

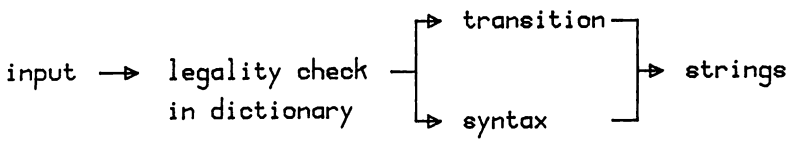


Figure II.1

With this model as a reference we will take a closer look at each element in the next sections, indicate the difficulties in implementing it and present the strategy we have chosen.

II.3. The dictionary

A player likes to talk to the computer in a more or less natural language. However, what is keyed in on the keyboard has to be interpreted and this is a very difficult job because of the ambiguities inherent in all natural languages. Example: boy scouts fish; it can mean (boy scouts) fish, but as well boy (scouts fish). In this example scouts can be an object or a verb.

Even with the help of a mainframe computer it is difficult to give a reasonable interpretation of phrases in some natural languages. Therefore on a microcomputer there will always be a bit of a compromise concerning such phrase-analysis. We will call the commands or words typed in by the player the *input* to the game. In most adventure games the input is of the form:

- some verb representing an action,
- some verb applied to an object.

Examples are: "north" or just "n", "open door", "take key", etc. In this book we will allow only these two kinds of inputs as commands (Other input is possible, but only allowed if requested by the game). This does not really restrict the pleasure a player can have in a game. A too restrictive dictionary however does. If nearly every word typed in by the player results in:
"I don't know this word"
you would get annoyed pretty quickly.

On the other hand a large dictionary makes the job of comparing the input with all the words known to the game very timeconsuming (and it takes up a lot of memory as well). In BASIC this often leads to ugly constructs like:

```
IF input$ = "take" GOTO 1000
IF input$ = "drop" GOTO 1100
      |          |
IF input$ = "north" GOTO 2000
PRINT "I dont know this word"
```

In our model the search through the dictionary is not by programming, but by a technique called hashing. For those who are interested in this technique we refer to section IV.8.

The result of the search is not word "x" has been recognised (or not), but instead a unique number is returned that represents the word "x"; this number was assigned to "x" when it was defined and entered in the dictionary. Words have unique numbers, except for those words considered equivalent, e.g. "north" and "n". This means that from the moment the input typed by the player has been converted into unique internal numbers, only numbers are handled by the game. This makes the internal description a lot simpler and the interpretation a lot quicker. This is usually called "lexical analysis".

II.4. The transition graph

This is no doubt the simplest part of the game. It is of the form:

```
<source location> ::= <move verb> <destination> |
                      <move verb> <destination> |
                            ..          ..      |
                            ..          ..      |
                      <move verb> <destination>
```

When a player types a move verb, then the only thing that needs checking is if this particular verb is in the list of possible transitions from the current location. If so the player is transported to the destination location, otherwise a message like "You can't go in that direction" is returned.

As an example consider a small transition graph with only four locations. First we give a representation that is easy to use when designing a game:

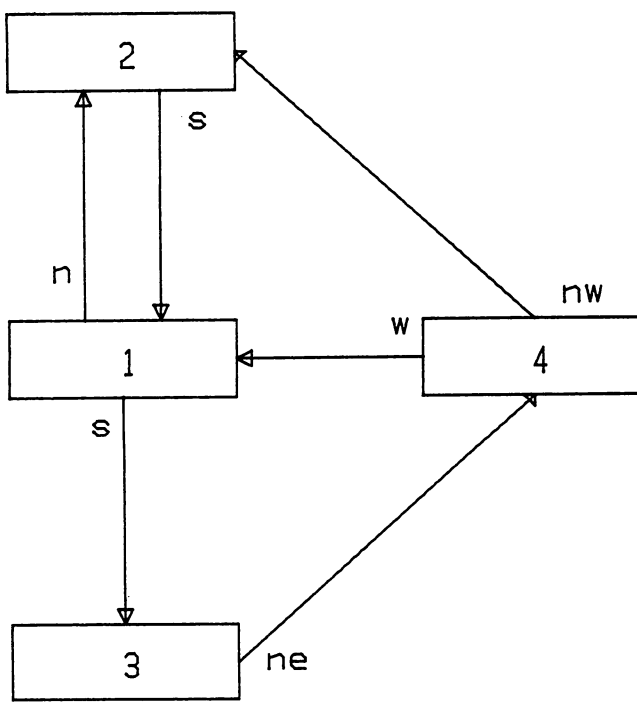


Figure II.2

In Figure II.2 you can go from location 1 north to location 2 and south to location 3. From location 2 you can go south to location 1 only. It is evident that there has to be some limit to the number of transitions from a place. A maximum number of five directions from any location is adequate, and allows very complex transition graphs. On the other hand this number still allows a player to make a map (with arrows) of the game. Too many transitions would make a knot of the arrows.

The map in Figure II.2 can also be represented by the following table:

current location	*move verb*	*destination*	*move verb*	*destination*
1	north	2	south	3
2	south	1		
3	ne	4		
4	nw	2	west	1

II.5. The strings

Clear descriptions of locations and objects and appropriate messages to the player are the basis of a good adventure. Such clarity is, however, expensive in memory requirements. Nevertheless the possibility of creating large adventure games on a small micro is kept open, often using very tricky compactions. A compaction is the encoding of text in such a way that it occupies less space than the original Ascii-coded text.

Section IV.7 describes a compaction scheme called Huffman encoding; this encoding saves up to 50 percent of the space the text normally takes. The adventure game writer however does not have to worry about this encoding; he can write his strings in the normal way. The encoding is only used internally. This will be explained in the following sections.

II.6. A special purpose adventure language

II.6.1. Introduction

We have discussed how a player can move around in an adventure and how this can be implemented. It is rather straightforward as long as you have a transition table and strings for the locations. Each time a transition occurs only the string belonging to the new location has to be printed, to tell the player where he is.

If you could only move around, however, you would lose interest pretty soon. The things that make an adventure game interesting are the puzzles the player has to solve; finding treasures, killing monsters etc. Some objects are aids to cope with other objects; the player has to locate these aids and use them.

In most adventures the manipulating of objects and the solving of puzzles is programmed in a computer language. On micro computers this is usually done in BASIC. There are several reasons why BASIC is not really suitable for writing adventure games:

- it is a general computer language and not specially designed for writing adventure games,
- it does not enable a game architect to write well structured programs; adventure games require a lot of bookkeeping and to implement them properly as much structure as possible is needed, not only for the first implementation, but also to allow changes and extensions to the program. An implementation in table form, like the transition table, would be ideal, but unfortunately the description of the puzzles cannot be structured to that extent. It would impose too many limitations and make the games dull,
- there is a lot of difference between the BASIC dialects on various computers. Writing a game on one type of micro does not guarantee that it will run on another one as well.

The possibility to use software, written for computer A, on computer B is called *portability*. The different BASIC dialects in use on different computers prevent portability of software written in BASIC to a large extent.

II.6.2. Why an adventure language ?

A special purpose adventure language can solve all these problems. Because

such a language is designed just for the purpose of writing adventure games, it facilitates the writing and thus the implementation of a game. This will make it possible to spend more time and attention on the creative aspect, which in turn will make it possible to produce better quality adventure games. Another advantage of a special purpose adventure language is, that adventure game writers need not know BASIC or the machine code of their computer; they can directly write games in the adventure language, so that adventure programs can be kept very simple.

Every feature of the language is concentrated on adventures. As a side effect it also delivers a much more structured approach; both because it is designed to do adventure bookkeeping and because of its simplicity. Finally it can also solve the problem of portability. We will touch on this point in the next section, where we discuss the implementation of the adventure language.

II.6.3. Adventure language implementation

A micro does not understand BASIC or machine code directly. At the machine level the processor operates on machine instructions, which consist of combinations of zeros and ones. These bit patterns form an encoding for the instructions; what a certain bit pattern means and how many bits are used depends on the type of processor (There are for example 8, 16, and 32 bits based micros).

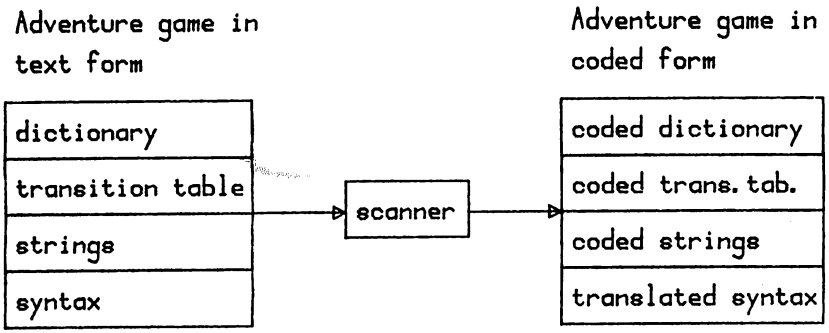
Programming a computer in machine code is very difficult; it is quite a task to remove bugs from machine code programs. Especially big programs are virtually impossible to create and to modify. This is because bit patterns may be easy to understand for a machine, but not for human beings. To diminish the distance between man and machine, computer languages have been created, as a more easy way to describe what you want the computer to do. BASIC is an example of such a language.

The BASIC statements have to be translated into machine code, which is done by a BASIC interpreter or compiler. This is a program in machine code and is usually stored in ROM. An interpreter reads a program line after line, interpreting and executing. This is done every time the BASIC program is used. A compiler on the other hand reads the program once and translates it into machine code. After the compilation this can be executed directly. The original program in BASIC is then called the source program and the compiled version the executable image. The BASIC interpreter also has to be resident if you want to use a BASIC program. Because an executable image is executed directly it is generally much faster than interpreted code (at least ten times).

The adventure language that will be described in the next sections is compiled. The program that reads and translates a program written in ADL (Adventure Description Language) is called 'the scanner'. This name (instead

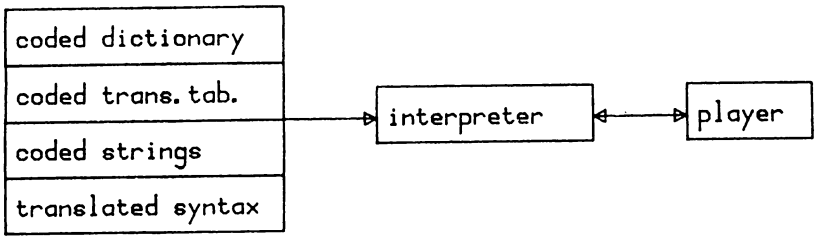
of compiler) has been chosen because the scanner performes a translation of the transition table and dictionary into an internal format as well. The scanner is a program written in BASIC and is different from a normal compiler, because it does not produce machine code; instead it produces a very compact special adventure code. This code cannot be executed directly; a second program is needed, 'the adventure code interpreter'. This is also a program written in BASIC. This interpretation can be compared with BASIC interpretation, with the difference that it was not a human being that wrote the input of the interpreter, but instead by a program (the scanner).

The scanner is used only to translate the adventure game into an internal code; each time someone likes to play, he has to use the interpreter. A full description of the files that are needed for the scanner to provide a complete adventure will be given in Chapter III. In Figures II.3 and II.4 we present a scheme to give you some idea.



Scanning.

Figure II.3



Playing.

Figure II.4

A BASIC program tells the BASIC interpreter what the programmer wants the computer to do. In the same way the coded file produced by the scanner tells the adventure game interpreter what he wants the adventure game to do.

The scanner provides as much help to the adventure writer as possible. For example, it performs the following checks:

- whether or not all locations in the transition table can be reached from at least one other location,
- whether or not the syntax for all verbs agrees with the specification of the adventure language.

The last point does not imply that it is possible to check whether all the puzzles are solvable, and whether it is possible to reach the final solution. This would mean more or less creation of adventure games by a computer; fortunately, the creative part is still in the hands of human beings.

II.7. Verbs and objects

The definition (or the ingredients) of ADL is based on our own experience with adventure games. This definition is by no means the only one possible. There are very many ADL's that can be created, similarly to normal programming languages and dialects. We have chosen a main set of ingredients with which, we think, most adventure games can be implemented.

We will start with some commands or phrases that can be typed by the player, and give an intuitive explanation of what the computer has to do to respond to these commands. The examples given below appear in most games. We do not touch upon commands that cause transitions because these are taken care of in the section about the transition table. How the distinction is made between the various groups of commands, i.e. move verbs, transitive and intransitive verbs and objects, will be explained in section II.9.

II.7.1. Intransitive verbs

This group is rather limited, because verbs without an object are in general not concerned with a specific action in an adventure; they usually provide information to the player. Some examples of verbs in this group are:

invent(ory), look, back and help

- invent : list the objects the player is carrying,
- look : list the current location,
- back : go back to the previous location. Another implementation might be: if the previous location is still known go back there, otherwise tell the player that you have lost all sense of direction,
- help : give a hint at those locations where help is avai lable or interesting, otherwise respond that there is no help for that location.

II.7.2. Transitive verbs

The verbs in this group cause changes in the game, and are therefore more interesting. Examples are not hard to find:

take, drop, and open

- take object : if the object is present at the current location move it to the players inventory, otherwise respond that you do not see the object here,
- drop object : if the player carries the object, i.e. it is in his inventory, move it to the current location otherwise respond that he is not carrying the object,
- open object : the object will be something specific e.g. door or gate. If there is no door or gate the response will be: ''there is nothing here to open'', other wise the player might need a key. If he has the right key then actually open the door, otherwise respond that he needs a key to do this.

The above examples show that some transitive verbs are applicable to all objects, while others are only applicable to specific objects. For example, in the case of ''take'' it would be useful if objects could be divided into groups, on one hand the objects that can be moved (that is, taken) and on the other hand the objects that can not. More about properties of objects in section II.7.4.

II.7.3. Conditional and random transitions

Suppose that at some location there is a way south, but it is blocked by a gate. Initially the gate is closed, and must be opened by the player. So it depends on the state of the gate whether or not the player is allowed to go south. This could be described as:

> if the gate is open, make the transition south,
> otherwise respond to the player that the gate is closed.

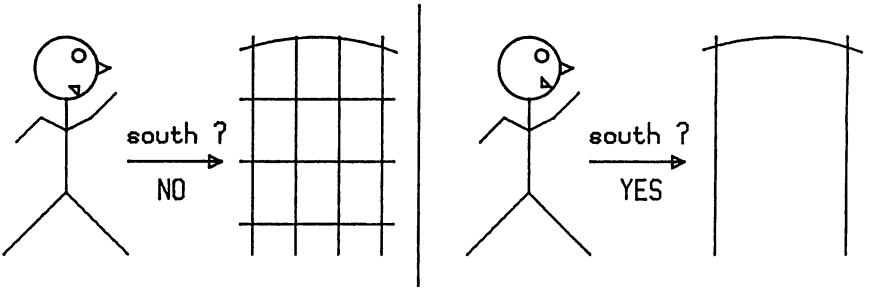


Figure II.5

This is called a *conditional transition,* the condition in the above case being that the gate has to be open.

On the other hand, suppose that a player is at a certain location, and typing exactly the same move verb leaves him either at location "x" or at location "y".
This is called a nondeterministic or *random transition:* instead of there being only one possible transition, both "x" and "y" have some probability to be selected, depending on how the game has been implemented.
Of course it is possible that there are more than two destination locations, and that the starting location is one of them (this trick is frequently used for creating a maze).

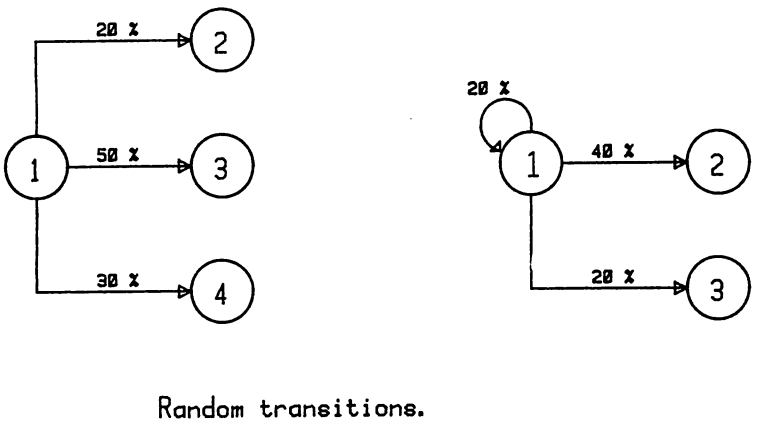


Figure II.6

II.7.4. Objects

Apart from the verbs and the commands of the game, objects are the basic ingredients for the puzzles. In our model the objects have several characteristics which originate from our experience with playing adventure games. Every object has:

- a class, which defines the type of the object,
- a location number, which defines where the object is at the moment,
- a string number, which points to the string describing the object.

The objects are divided according to their class into groups. Some examples are "FRIEND", "ENEMY", "AID" and "TREASURE". During the adventure

the above characteristics are often examined and changed according to actions of the player. For example "kill object" may result in removing the object from the game, or in changing its description and class.
"Take object" could only be allowed for "AIDs" and "TREASUREs", but certainly not for "FRIENDs" and "ENEMYs". How this is implemented in ADL will be described in the next section.

This completes our description of the classes in the dictionary.

We now have:

- move verbs (normal, conditional or random)
- transitive verbs
- intransitive verbs
- objects

II.7.5. Other ingredients

We are almost ready for the definition of the Adventure Description Language. There are several practical addenda which need to be used for describing words, some of which are necessary, while others just facilitate the work of the adventure writer. These addenda are only used when a game is written; the player remains totally unaware of them:

- *variables:* just like other programming languages
  ADL has variables. There is only one type: integer. The variables are used to do the necessary bookkeeping of the actions performed by the player. Five variables have been predefined:

  -CURLOC-the current location of the player
  -OLDLOC-the previous location of the player
  -SCORE-the score
  -NTURNS-the number of turns
  -DUMMY-a scratch variable for temporary use

- *dereferencing:*
  dereferencing delivers the value of a variable. The operator for dereferencing is the character "?". Dereferencing is only applicable to varia bles, and the resulting type is integer. E.g. CURLOC means the variable that stores the current location, ?CURLOC is the value of the variable CURLOC, i.e. the number of the current location.
- *object condition:*
  in order to simplify the description of verbs, an object condition has been introduced which has to be satisfied before the verb is executed. There are three possibilities:

-LOCAL, the object has to be at the current location,
-INVENT, the object has to be carried,
-SKIP, no special condition.

e.g. ''take object'' would normally be described as:
if the object is at the current location then....
in ADL, this will become ''take object LOCAL .....''
Another example is: ''drop object INVENT .....''

- Finally two general words have been added:

-CLAY this matches any object typed by the player,
-ALLMOV this matches any move verb typed by the player.

II.8. The Adventure Description Language

II.8.1. Introduction

We will now introduce the adventure language. The constructions in the language are commonly found in most existing programming languages and therefore they pose no problems to a reader with some programming experience. But to inexperienced programmers they will be quite straightforward as well, because the constructions shown are very similar to the intuitive descriptions used in II.7 for verbs. For those who have problems with the formal description, the examples in section II.8.8 give some applications.

II.8.2. Main syntax constructions

Because we can have verbs with and without objects, there are also two main syntax constructions:

```
-  <transitive verb> OBJECT-1 <object condition> <statement>
     NEXT            OBJECT-2 <object condition> <statement>
     NEXT                |                |            |
     NEXT            OBJECT-n <object condition> <statement>
     END
                     ( with n > = 1 )

-  <other verb>      <statement>
     END
```

The above lines are called the *syntax rules* of the adventure language. The first rule merely says that a transitive verb must be applied to at least one object, but can be applied to different objects. Every object has its own condition as mentioned in II.7.5 and <statement> is the description of this particular combination. The pair of brackets <..> around an identifier indicate that the syntax description is not yet finished, but instead a description of that item will follow. For example the syntax rule for <object condition> is:

<object condition> :: = LOCAL | INVENT | SKIP

The "|" sign is a separator to indicate alternatives. The above rule states that <object condition> equals LOCAL or INVENT or SKIP.

The word NEXT is used to indicate that this particular verb is applicable to yet another object. At the word END the description of the verb is finished and a next verb can follow.

Complete examples of syntax constructions of the adventure description language are:

1) *The transitive verb "open":*

```
open door LOCAL   <statement describing what to do to open the door;
                  does the player have a key etc....>
NEXT gate LOCAL   <the same but now a statement describing what
                  to do to open the gate.....>
END
```

(LOCAL means that the door and the gate have to be present at the current location to enable "open" to be executed)

2) *The intransitive verb "invent":*

```
invent            <statement to list the items carried by the player>
END
```

II.8.3. The statement

We will now explain what a statement is. Here is its syntax rule:

```
<statement> ::= <statement part> <statement> |
                <statement part>
```

For <statement> you can make your choice from two alternatives: either <statement part> <statement> or just <statement part>. In the first alternative <statement> also appears at the right hand side; this is called a recursive definition. In chapter IV we shall say more about this way of defining a syntax for those who are interested. Right now it is enough to know that a <statement> is a list of one or more <statement part>s; what a <statement part> is, is stated by the following syntax rule:

```
<statement part> ::= <if statement>                    |
                     <case statement>   |
                     <action procedure>
```

These three alternatives are discussed in the following sections.

II.8.4. The IF statement

Again we will first present the syntax rule for an if statement and give its explanation afterwards. To avoid making things too complex we will not give the formal definition, but a simpler one. The formal definition can be found in Appendix A.

```
<if statement> :: =
IF <condition> THEN <statement>
{ELIF <condition> THEN <statement>} (zero, one or more times)
{ELSE <statement                    > } (optional)
 FI
```

Note that the last four lines do not indicate alternatives; all of them are part of the if statement. The first and the last line must be present at all times. The ELIF part (line) may be absent, and if it is present it may occur more than once. The ELSE part is optional. Words like IF, THEN, ELIF, ELSE and FI appear in the text of the adventure language writer and are called *reserved words* of the syntax. Other reserved words we have met so far are CLAY, ALLMOV, LOCAL, INVENT and SKIP.

The *execution* of an <if statement> is done in the following way: the condition is evaluated and results in the boolean value TRUE or FALSE (for more on boolean values see BBC User Guide, Chapter 16). If the condition yields TRUE, the statement after the THEN is executed, and the rest is skipped until the FI (indicating the end of the <if statement>).
Otherwise the condition of the first ELIF (if present) is evaluated and likewise when it is TRUE the statement of the corresponding THEN is executed. If none of the conditions was TRUE and an ELSE part is present, the statement after the ELSE is executed. If it is not present the <if statement> has no effect at all.

```
IF EQUAL STATE 0 THEN actions1
ELIF EQUAL STATE 1 THEN actions2
ELIF EQUAL STATE 2 THEN actions3
ELSE actions4
FI
```

value of state:	executed actions:
0	actions1
1	actions2
2	actions3
other values	actions4

The only part of the <if statement> that remains undefined is the <condition>, which will now be discussed.

II.8.5. Conditions

When the result of a <condition> is evaluated, its result is always TRUE or FALSE. The syntax rule for a <condition> is:

```
<condition> ::= <simple condition> AND <condition> |
<simple condition> OR <condition> |
<simple condition>
```

Again a recursive definition with three alternatives. The rule for <simple condition> is:

```
<simple condition> ::= NOT <condition function> |
                       <condition function>
```

This rule states that a simple condition is a condition function with or without a NOT in front of it. A condition is a list of simple conditions separated by OR's and AND's. The function of the reserved words NOT, AND and OR is simple, if you consider that they are only applicable to the values TRUE and FALSE (calculation with booleans is called boolean algebra).

A NOT reverses a boolean value: NOT TRUE = FALSE and
NOT FALSE = TRUE
An AND between two boolean values results in:
TRUE AND TRUE = TRUE
TRUE AND FALSE = FALSE
FALSE AND TRUE = FALSE
FALSE AND FALSE = FALSE
Likewise for the OR:
TRUE OR TRUE = TRUE
TRUE OR FALSE = TRUE
FALSE OR TRUE = TRUE
FALSE OR FALSE = FALSE

Finally the <condition function> forms the basic element of the §condition>. It will be discussed in full detail in section II.8.7.

II.8.6. The CASE statement

A case statement has the following syntax rule:

```
<case statement> ::= CASE      variable
                     OF        values-1 : statement
                     OF        values-2 : statement
                     ..           ..    :    ..
                     OF        values-N : statement
                     DEFAULT             statement
                     ESAC
```

The OF-part must be present one or more times, but the DEFAULTpart is optional. The values-i are lists of one or more values; for example, "2,3,5,13".

When this statement is executed, the value of the variable at the moment of execution is compared with the values of the different OF-parts. If they are equal the statement of that OF-part is executed and completes the CASE statement. If none of the values match and a DEFAULT-part is present then the statement of the DEFAULT-part is executed. The ESAC indicates the end of the CASE, just like the FI indicates the end of the IF statement.

A CASE statement can also be simulated by an IF statement, but the result is rather obscure.

```
IF    var = values-1 [ 1 ] OR var = values – 1 [ 2 ] .... THEN OF__part__1
ELIF  var = values – 2 [ 1 ] OR var = ....                 THEN OF__part__2
  |    |          |              |          |                  |          |
ELIF  var = values – N [ 1 ] OR var = ....                 THEN OF__part__N
ELSE  DEFAULT__part
FI
```

II.8.7. Action procedures and Condition functions

The action procedures and condition functions are the last items of the syntax that remain undefined. They form the basic building blocks of adventure creation. Action procedures are similar to procedures in programming languages, and condition functions to functions. From now on we will abbreviate with "actions" and "conditions".

An action is a preprogrammed (therefore predefined) element, which causes changes in the bookkeeping of the adventure game, and is used to administer the player's whereabouts. We have tried to provide a set of different actions which would be sufficient to implement a wide variety of games. These are listed in Figure II.7.

The conditions differ from the actions, in that they are used to find out in what kind of situation the player is in. A condition returns a boolean value and can be used in an IF statement to decide which statements are to be executed next.

Both actions and conditions have an identifier which defines which particular action or condition must be executed, and a list of parameters. In our definition there may be one or two parameters, or no parameters at all.

Every parameter has a certain type, like LOCATION, NUMBER or VARIABLE. The scanner checks both the number and the type of these parameters. Thus in the following statement:

```
MESSAGE CURLOC
```

Figure II.7

```
CHLOC (OBJ, LOC)        : Change the location of object
                          OBJ to LOC
                          (location 0 is the inventory
                          of the player)
CHCLASS (OBJ, CLASS)    : Change the class of object
                          OBJ to CLASS
CHOSTR (OBJ, STRNR)     : Change the string of object
                          OBJ to STRNR
SETRAN (VAR, VAL)       : Assign to variable VAR a
                          random value between 0 and
                          VAL
SETVAR (VAR, VAL)       : Assign value VAL to variable
                          VAR
ADD (VAR, VAL)          : Add value VAL to variable VAR
SUB (VAR, VAL)          : Subtract value VAL to
                          variable VAR
PRINT (VAR)             : Display the value of variable
                          VAR
READI (VAR)             : Read an integer value from
                          the keyboard and store it in
                          variable VAR
REPROC (VERB, OBJ)      : Execute verb VERB with object
                          OBJ
LISTLOC (LOC)           : List location LOC
MESSAGE (STRNR)         : Print the message with number
                          STRNR
SLEEP (VAL)             : Wait VAL seconds
PRINTA (ASC, NR)        : Print ascii character ASC NR
                          times
QUIT                    : Exit the adventure game
CHLSTR (LOC, STRNR)     : Change the string number of
                          location LOC to STRNR
ALARM (VERB, NR)        : Execute verb VERB after NR
                          turns of the player
```

the scanner would complain that a message number was expected and not a variable. In general the scanner will provide as much help as possible in checking the consistency of the adventure program. This will save a lot of time and effort.

All predefined conditions are listed in Figure II.8. Note that only SMALLER and SMEQUAL are defined and not GREATER and GREATER-EQUAL, since these can be programmed using a NOT:

```
NOT SMALLER = GREATER-EQUAL
NOT SMEQUAL = GREATER
```

Figure II.8

```
PRESENT (OBJ)          : Test whether an object OBJ is
                         present at the current
                         location
HOLDING (OBJ)          : Test whether the player is
                         carrying object OBJ
TESTCL (OBJ, CLASS)    : Test whether the class of
                         object OBJ is equal to CLASS
EQUAL (VAR, VAL)       : Test whether variable VAR has
                         value VAL
SMALLER (VAR, VAL)     : Test whether variable VAR has
                         value smaller then VAL
SMEQUAL (VAR, VAL)     : Test whether variable VAR has
                         value smaller or equal to VAL
READB                    : Read yes or no from the
                         player
OBJSTR (OBJ, NR)       : Test whether object OBJ has
                         string number NR
LASTOBJ (OBJ)          : Test whether the last object
                         typed by the player is OBJ
OBJLOC (OBJ, LOC)      : Test whether object OBJ is at
                         location LOC
```

II.8.8. Examples

Out of the maximum possible number of locations, two are reserved:

–LOCATION 0	is the inventory,
–LOCATION maxloc	is the store room of the adventure; here the objects that are not in the game are being stored.

Figure II.9

```
invent      LISTLOC 0
            END
```

If the verb "invent" is typed by the player, a list of what he currently holds has to be printed. In ADL this can be programmed with one statement as is shown in Figure II.9. The action LISTLOC lists a location. It has as a parameter location number 0, which is defined (see above) as the inventory.

Figure II.10

```
look        LISTLOC ?CURLOC
            END
```

'Look' means that the current location has to be listed. The current location is stored in the (predefined) variable CURLOC. The parameter of LISTLOC is a location number. With the (dereferencing-) symbol "?" the value of the variable CURLOC is provided to LISTLOC, and the current location is listed.

Figure II.11

```
back    IF EQUAL CURLOC ?OLDLOC
        THEN MESSAGE x
        ELSE SETVAR CURLOC ?OLDLOC
             LISTLOC ?CURLOC
        FI
        END
```

If the variables CURLOC and OLDLOC have the same value (that is the player is at the same place as one turn ago), print a message with message number "x".
This message could be something like:
"I have lost every sense of direction"
If there is no equality then execute the ELSE part: Assign the value of the variable OLDLOC to variable CURLOC. This will put the player back and list the location (see Figure II.10).

Figure II.12

```
take    CLAY LOCAL   IF EQUAL NCARRY 5 THEN
                     MESSAGE y
                     ELSE CHLOC CLAY 0
                          ADD NCARRY 1
                          MESSAGE z
                     FI
        END

message y:   "You are carrying too much, please drop something first."
message z:   "OK !"
```

CLAY is a reserved word indicating any object. The object typed by the user is substituted for CLAY and "take" acts on it. The object has to be LOCAL; if it is not present at the current location the IF-statement will not be executed. If the variable NCARRY (indicating the number of objects the player is carrying) equals 5, then refuse to pick up any new objects. Otherwise change the location of the indicated object to zero (the inventory) and record this fact by adding one to the variable NCARRY. Finally tell him that his action was successful ("OK !").

Figure II.13

```
drop        CLAY INVENT  CHLOC CLAY ?CURLOC
                         SUB NCARRY 1
                         MESSAGE z
            END
```

Again CLAY defines any object known to the game. If "drop" is entered the object has to have the property INVENT, that is it must be in the inventory. If so the location of the object is changed to the value of CURLOC, in other words it is transferred to the current location. We must update the variable NCARRY (by decrementing it by 1) to indicate the fact that the player carries one object less. Again the player is notified that his action was successful.

Figure II.14

```
help        CASE            CURLOC
            OF              3 : MESSAGE a
            OF              5 : MESSAGE b
            DEFAULT MESSAGE c
            ESAC
  END

message a:   "I would not go west here"
message b:
             "You can get a key at the reception"
message c:   "Sorry, no help info for this location"
```

It will depend on the value of the variable CURLOC which message will appear on the screen.

II.9. Adventure game implementation using ADL

The adventure game writer has to create four text files, which together form a complete adventure game. Because these files are normal text files they can be created using any editor. For the adventure game writing on the BBC a screen oriented editor (or display editor) has been added to this book in Appendix E.

We have already indicated in previous sections what these four files have to contain: a dictionary, a transition table, a file with strings and finally a syntactic meaning of the actions, written in the Adventure Description Language. In this section we will describe these in more detail. The scanner will also be explained in more detail. In particular the checks it performs and what the coded adventure game file (produced by the scanner) looks like will be described. In the last subsection an explanation will be given of the interpreter. The use of the different coded tables and the translated syntax will be developed.

II.9.1. The dictionary

The dictionary is no more than an enumeration of the words recognised by the interpreter.

The dictionary consists of 5 sections: variables, move verbs, intransitive verbs, transitive verbs and objects. The first section, which contains a list of all variables, is used by the adventure game writer for bookkeeping and is therefore only for internal use. The other sections are coded by the scanner and entered in a coded dictionary, for use by the interpreter in recognising legal input from the player. The words are not entered sequentially (which would require a search from the beginning to the end through the dictionary to find some specific word) but instead are entered by a technique called *hashing*. The index in the table where the word is stored is calculated according to some algorithm which uses the Ascii values of the characters of the word. The same algorithm is used by the interpreter and a word can thus be found in most cases in the dictionary by comparing it just once. Only in the cases in which several words result in the same index two or more comparisons will have to be made.

This technique allows a very rapid search, even in large dictionaries. To avoid too many collisions (words with the same calculated index), the table must be about 15% bigger than needed. That is also the disadvantage; it costs some extra memory.

In Figure II.15 a sample dictionary has been given; it is advisable to put every word on a separate line, and to write the variables in upper case.

The variables do not have to be assigned a number in the dictionary, because they will be numbered sequentially beginning at the value 6. Remember that the first 5 variables are predefined (section II.7.5).
In this example the variable DOOROPEN is defined by the adventure game writer, to store the state of the object "door". It can be manipulated for example by the actions "SETVAR DOOROPEN value", or "CHVAR DOOROPEN value". The initial value 0 could mean the door is closed, and the value 1 could mean the door is open. The first section ends with a $ sign on the first position of the line. The rest of the line can be used as a comment (it wll be ignored by the scanner).

In the second section (the move verbs), it becomes evident why these words do have an identification number in the dictionary; it would be impossible to define different words with the same meaning. The words "n" and "north" both have identification 1, which during the game will result in the same action.

Figure II.15

```
DOOROPEN
[
n          1
north      1
s          2
south      2
w          3
e          4
u          5
d          6
[
look       1
invent     2
examine    3
[
take       1
drop       2
open       3
close      4
lock       4
[
a door     1  NOCLASS  4    1
a key      2  AID      6    2
an apple   3  AID      5    3
[
```

You are allowed to use the same identification numbers in one section of the dictionary as in another one (for instance north = 1 in section two and look = 1 in section three). The scanner transforms the numbers, allowing the interpreter to determine to which class a word belongs.
If you use the same number for different words in one section you indicate that these words have to be considered equal as in the example of "n" and "north". Again the section is terminated by a $ sign.

Sections three and four are the intransitive and the transitive verbs and they are treated just like the move verbs.

Finally section five defines the objects. More information has to be provided for an object, to indicate its properties. The particle in front of the object is used by the interpreter for the generation of messages. The first number is the identification number just like in the other sections.
The next item is the CLASS of the object. In our case "door" has class NOCLASS or in other words no special properties. The other object "key" has class AID. This property can for example be used in the verb "take" to distinguish between the facts, that the player can take the key but not the door.
Finally the two last numbers indicate the location where the object (initially) is, and the number of its object string. The class, location number and string number are all properties of an object, and they can be modified during the play (the values in the dictionary are the initial values). Note that every section in the dictionary is closed with a $ sign.

II.9.2. The transition table

Because of its simplicity we already presented the structure of the transition table in section II.4. Again it is wise to put all the transitions from one location on a single line. Although you are not obliged to do so, the lines in the table should be placed in ascending order. This makes it easier for the writer to find a location and its transitions. Another example of a transition table has been given in Figure II.16:

Figure II.16

```
1 in     2
2 out    1   n   3   s   4
3 d      5   s   2
4 n      2   e   0
5 u      3   s   4   d   6
6 u      5
7 w      0
```

In Figure II.17 you can find the equivalent map representation of this table. When you are creating a game such a map is much easier to write down. First make a map and then transform to text form.

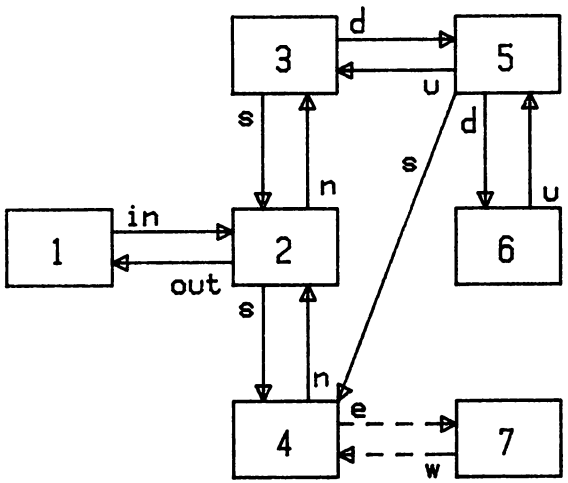


Figure II.17

Two transitions, those from 4 to 7 and from 7 to 4, are special. These read: e 0 and w 0. The 0 indicates that the transition is special. It is described in the syntax and is conditional or random (it does not imply a transition to location 0!).

II.9.3. The strings

The string file contains 3 sections all with the same structure:

- a "#" character at the first position of the line,
- a string number in the range 0..255,
- a string.

The sections are separated by a $ sign at the first position of the line. This is similar to the way sections are separated in the dictionary. The sections are: general messages, descriptions of locations and descriptions of objects. If a string is split over several lines, it will appear on the screen in the same format. An example of a string file has been given in Figure II.18.

The first four strings are general messages.
The next seven strings are the descriptions of the locations in the transition table of Figure II.17. The last four strings are object descriptions for the key, door and apple in the dictionary in Figure II.15.

Figure II.18

```
#1   OK !
#2   You are not carrying a key.
#3   The door is already open.
#4   The door is already closed.
#5   You bump your head against the closed door!
$    end of messages
#0   You are currently holding:
#1   You are in front of a building. The only way is
     in.
#2   You are inside the building.
     Passages lead north and south.
#3   You are in the northern room, there is a
     stairway down.
#4   You are in the southern room.
#5   You are on top of a pit. A stairway is leading up
     and there is a corridor to the south.
#6   You are in a pit.
#7   This is the special room.
$    end of locations strings
#1   There is a closed door in the corner.
#2   There is a key on the ground.
#3   A fine red apple lies before you.
#4   In the corner is an open door.
```

II.9.4. The syntax for verbs

The form of the syntax constructions has been described in section II.8. The syntax file consists of a syntax construction for each transitive, intransitive and special move verb. In figure II.19 an example syntax is shown describing the verbs in the previous examples.

The syntax for "east" and "west" covers the conditional transition shown in section II.9.2. If the door is open (if the variable DOOROPEN is equal to 1) then make the transition from location four to location seven or vice versa, otherwise print a message indicating that the door is closed.

Figure II.19

```
e     IF EQUAL DOOROPEN 1 THEN CHVAR OLDLOC 4
                               CHVAR CURLOC 7
                          ELSE MESSAGE 5
      FI
END
w     IF EQUAL DOOROPEN 1 THEN CHVAR OLDLOC 7
                               CHVAR CURLOC 4
                          ELSE MESSAGE 5
      FI
END
take key LOCAL
      CHLOC key 0
      MESSAGE 1
END
drop key INVENT
      CHLOC key ?CURLOC
      MESSAGE 1
END
open door LOCAL
      IF EQUAL DOOROPEN 1 THEN MESSAGE 3
      ELIF HOLDING key THEN SETVAR DOOROPEN 1
                            CHOBJSTR door 64
                       ELSE MESSAGE 2
      FI
END
close door LOCAL
      IF EQUAL DOOROPEN 0 THEN MESSAGE 4
      ELIF HOLDING key THEN SETVAR DOOROPEN 0
                            CHOBJSTR door 1
                       ELSE MESSAGE 2
      FI
END
```

''Take'' and ''drop'' in this example only apply to ''key''. The interpreter will reply to ''door'' something like: ''I can't apply this verb to this object''. For ''take'' the object condition LOCAL guarantees that the key has to be present at the current location before the statement is executed. The statement ''CHLOC key 0'' will change the location of the key to 0 or the inventory. Likewise at drop INVENT guarantees execution only in the case that the key is in the inventory. If so ''CHLOC key ?CURLOC'' will change the location of the key to the contents of the variable CURLOC (the current location).

In the case of ''open'' and ''close'' a test is first made whether or not the door is already open or closed in which case a message is printed; otherwise it is tested whether or not the player has a key. If he does not, the player is

informed that he will need a key, otherwise the door is opened or closed. In the latter case the interpreter will change the value of the variable DOOROPEN (which contains the state of the door) and it will also change the string describing the door.

II.9.5. The scanner

The last file read by the scanner is the file containing the strings. First, the scanner will check that three sections are present. Then the strings are converted to Huffman coded bit patterns (see section IV.7). Also a table of the memory locations of the start of these strings is constructed.

The first file read by the scanner is the dictionary. A check is made that exactly five sections exist (a section might be empty with the exception of the "$" sign), and whether or not the contents of these sections are correct. Two tables are constructed:

- a table with reserved words for internal use,
- a coded dictionary for the interpreter.

The variables are stored in the first table. Note that the scanner adds the five predefined variables and the reserved words. These reserved words are:

- the reserved words in the syntax: IF, CASE, NEXT etc.,
- the classes of the objects: AID, PERSON, NOCLASS etc.,
- the action identifiers: CHLOC, REPROC etc.,
- the condition identifiers: TESTCL, EQUAL etc.

All words which can be used by the adventurer are stored in the coded dictionary: the move verbs, intransitive and transitive verbs and objects. The properties of each object are stored in another table.

The next file is the transition table. For each location in the table a check is made whether or not each verb used to define transitions from that location is present in the dictionary, and whether these were introduced in the dictionary as a move verb.

A coded transition table is generated in which the move verbs are replaced by their unique identification numbers. Finally a cross check is made, in which any isolated locations (locations unreachable from other locations) are detected, and finds locations for which no list of transitions has been defined. In the first case a warning is printed. It is not possible to make a full check, as there may exist conditional and random transitions from or to a location. The second case is fatal, however, because if a player enters an undefined location the bookkeeping would no longer be correct. If you want to create a location which a player cannot escape from you must use a special kind of

transition, ALLMOV, e.g.

```
20    ALLMOV 20
```

The next file read is the syntax file. Each verb in the dictionary (which is not a move verb) must have a syntax construction. On the other hand each verb and object in the syntax has to be present in the dictionary. Transitive verbs in the syntax require an object and an object condition, while other verbs are not allowed to have one.

The statements of each verb or (verb,object) combination are checked with for syntactic and semantic correctness:

- the syntax has to be correct according to the specification of ADL,
- the semantics of the ACTIONs and CONDITIONs has to be correct.

The latter case means that the number of parameters has to be correct, and that each parameter has to be of an appropriate type. For certain types of parameters a more detailed check can be performed; for example in the case of "CHLOC object location", the location number has to refer to an existing location in the transition table. Finally a table is created for the syntax code, in which the start index of each verb is stored.

If a fatal error occurs the scanner does not produce a coded adventure. This is because such an adventure would present nasty problems to the interpreter. You might wonder if it is still possible to produce a coded file, because so many checks are made by the scanner. But like the error checks made by a compiler for a high level programming language like Pascal, it will be a great help to you if you are made aware of your errors before you start playing. The scanner can locate a great deal of these errors for you, and if you remove them the chance that something strange may happen in your adventure bookkeeping is much smaller. Especially in very big adventure games it can be quite awkward and time consuming to debug hundreds or even over a thousand lines of BASIC. Instead of having to find all bugs by yourself, a program (the scanner) finds out why the game will not behave properly. You can also correct these errors quickly, because the scanner gives their exact location.

All the tables, the coded dictionary and strings, and the translated syntax are written to two files which form the coded adventure. After this the interpreter can be invoked to play the game.

II.9.6. The interpreter

When the interpreter is started it reads the coded adventure game. It initialises amongst others the variables and starts the game. Then the description of location one is displayed.

The player can now type commands which are read in and interpreted. He is allowed to type as many objects and verbs on a single line as there is space (currently 85 characters maximum).
If a word is not in the dictionary the player is warned accordingly by the interpreter, otherwise the word is converted to its unique internal number. From that moment only that unique number is used by the interpreter.

If an object is encountered, a check is made whether the last typed verb was a transitive verb and the combination (verb, object) is executed. Otherwise the interpreter asks the player what it has to do with the object.

In the case of a move verb, the transition table is searched to see whether the verb is in the list for the current location. If not the player is told that he cannot go in that direction. In the case of a normal transition the current and old locations (in the variables CURLOC and OLDLOC) are changed to a value obtained from the table and the new current location is displayed with all the objects present. In the case of conditional and random transition the table entry is equal to zero. In this case the processing is done just like transitive and intransitive words: by starting at the syntax entry of the verb and interpreting the statement which describes it.

In the case of transitive verbs a check is first made whether the (verb,object) combination is possible, before executing the statement. If not, the message "Impossible" is printed. If the object does appear in the list of the verb or if the verb can be used with CLAY (any object), the object condition is verified. If the object has to be LOCAL, but is not present at the current location, the interpreter responds: "I don't see a(n) <object> here". If the object has to be INVENT and is not in the possession of the player, the interpreter responds: "You are not carrying a(n) §object>".

The execution of the statement generates messages or changes the bookkeeping. How detailed the response of the game is depends on the creator of the game, and on how clear he wants the game to be.

II.9.7. Conclusion

Although the abstract definition of an adventure, combined with several tables, seems complex, its use is not. The adventure game writer only has to provide the four text files and need not be aware of all the conversions, coding and table production performed by the scanner.

In chapter III we will build a small game step by step, and thereby show that adventure games can be written quite easily; indeed, they can be constructed bit by bit in a structured way. A game can also be extended and modified very easily; very big games can be divided into several manageable parts.

You do not need to know anything about the software and hardware of your

computer. The Adventure Description Language is a powerful tool which removes the need to program even one statement of BASIC or machine code. The language itself contains only a few conceptual elements and it is very easy to learn. The scanner provides as much help to the adventure game writer as possible. It runs a consistency check and most basic errors in the bookkeeping can be detected by it.

When one game has been implemented several general verbs like: "look", "inventory", "take", "drop" etc. can be reused in other games, because the meaning of these verbs is more or less the same in most adventure games. Also small extensions or adaptations can be made.

If you want to transfer games from one computer, (for example your own), to another, you need to rewrite the interpreter and the scanner for the other machine. This however does require knowledge of its software and hardware. From then on every game written on the first computer will also run on the second one and vice versa.

The only problem occurs when the definition of the Adventure Description Language is different on both computers. We do however think that the grammar of the language does not need to be adapted, because although it is simple it covers just about everything you need. The action procedures and condition functions might be extended; this hardly affects the language and the scanner, although it will require the addition of some small pieces of code to the interpreter.

In the next chapter we will present some theoretical background on grammars and languages.

CHAPTER III

The red button: a small game written in ADL

III.1. Introduction

In this chapter a description of a small game will be given in full detail. The description will be given along with the development of the game. First you must have a general idea for the final puzzle the player has to solve, then you can put in more puzzles and details which the player has to solve in advance.

The adventure programming system described in chapter 2 allows you to use this stepwise approach in a very easy way. It is also possible to test your partially developed game right away. This makes the creating process much more fun.

The game described in this chapter is certainly not the most original one you have ever seen, but everything had to be kept simple and small to enable us to describe it in full detail.

III.2. The basic idea

The basic idea of the game is that some lunatic has pushed the red button, and the only one that can stop the full automatic missile system is the player. To achieve this you have to go to the central computer, key in and shut down the computer system. The first step you, the Adventure Game writer have to take, is to draw a map describing the scenery of the game. The Computing Centre of the University where one of the authors works has served as a model for the map.

The next step is to complicate the keying in procedure and this requires the player to perform certain acts before he is able to shut down the system. This might require the creation of more locations on the map, and the addition of more transitions. For simplicity the total map (containing only 12 locations) will be shown straight away. Extending the number of locations is very simple: it only requires extending the transition table and adding strings for the new locations.

III.3. The map

In chapter II we already mentioned that the easiest way to make a map is to draw boxes representing the locations, and to indicate the possible transitions with arrows. The boxes are numbered, beginning with the starting location having number one. It also helps to give each location on the map a short name.

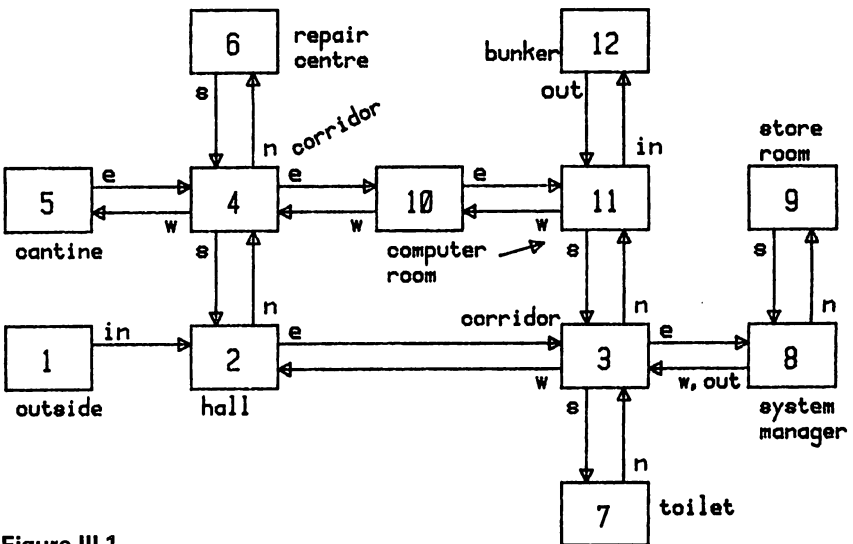


Figure III.1

The next step is to transform this picture to the form required by the scanner. Figure III.2 shows the equivalent transition table.

Figure III.2

```
 1   in  2
 2   e   3   n   4
 3   w   2   s   7   e   8   n   11
 4   w   5   n   6   s   2   e   10
 5   e   4
 6   s   4
 7   n   3
 8   w   3   n   9   out 3
 9   s   8
10   w   4   e  11
11   s   3   in 11   w  10
12   out 11
```

The short names of the locations are now extended to the string describing the locations (see III.3).

Figure III.3

```
#1  You are in front of a building reading ''Hexagon
    Computer Centre''. Some lunatic has pushed the
    red button and all systems run automatically.
    The only one to stop this madness is you....
#2  You are in the hall of the computer centre. To the
    north and east are corridors.
#3  You're in a corridor, you can go in several
    directions here. The sound of computer
    equipment can be heard.
#4  You're in a corridor, you can go in several
    directions here. The sound of computer
    equipment can be heard.
#5  You are in the canteen.
#6  You are in the maintenance room.
#7  You are in the men's toilet.
#8  You are in the room of the system manager.
#9  This is a small dusty supply room.
#10 You are in the computer room.
#11 You are in the computer room. In the corner is a
    little door.
#   The main terminal is here.
#12 You are in a little secret bunker.
```

In order to test this basic scenery we now provide a small dictionary to the scanner. The first words are extracted from the transition table: they are move verbs. The result is shown in Figure III.4.

Figure III.4

```
e      1
east   1
s      2
south  2
w      3
west   3
n      4
north  4
in     5
out    6
```

We now create an empty syntax file, add 2 empty string sections to the string file of Figure III.3 and add to the dictionaryempty sections for variables, transitive, intransitive verbs, and objects. You are now ready to produce your first coded adventure. Try to walk around according to the map; in this way you can check whether or not the descriptions of the locations are consistent with the directions as defined in the transition table.

III.4. The first puzzle

First we add to the game the process of keying in at the main terminal and running a shut down operation. For keying in we will supply the command ''login'' and for shutting the system the command ''run''. Both verbs will be taken intransitive and are added to the dictionary in the appropriate section. We will now present the syntax for both commands and add this to the syntax file:

```
login IF NOT EQUAL CURLOC 10 THEN MESSAGE 1
      ELIF EQUAL LOGIN 1 THEN MESSAGE 3
      ELSE SETVAR LOGIN 1
           MESSAGE 2
      FI
END
run   IF NOT EQUAL CURLOC 10 THEN MESSAGE 1
      ELIF NOT EQUAL LOGIN 1 THEN MESSAGE 4
      ELSE MESSAGE 5
           QUIT
      FI
END
```

variables: LOGIN intransitive verbs: login 1
run 2 messages: #1 I don't see a terminal here
#2 Welcome to Impossible Big Machine,
Chess is removed due to performance degradation.
#3 MCL—> Already logged in
#4 MCL—> Not logged in
#5 You saved the world, congratulations.

Explanation:

In both commands a terminal is required, and there is only one at location 10; at other locations message 1 will be printed. If the player is at location 10 the fact that he has used the command ''login'' is remembered in the variable LOGIN (which is part of the internal bookkeeping). If the player uses ''login'' more then once he gets message 3 (the ELIF part). Only the first time he is informed that he suc ceeded, with message 2 (the ELSE part). At the command ''run'' we test (the ELIF) whether the player has used the command ''login''; if not, he gets message 4. Otherwise he gets message 5 and the game is over.

Add the five messages to the string file, the words ''run'' and ''login'', the variable ''LOGIN'' to the dictionary, and the syntax rules to the syntax file. Again invoke the scanner to create a new coded adventure, and test this first puzzle. This process can be repeated after each extension to the game, and this way the development of the adventure can be checked systematically.

III.5. Extending the login procedure

In the previous section we gave a straightforward description of the "login" command. We will now extend this command, by changing its syntax. This will illustrate how easy it is to adapt and extend a game.

On most multi-user computer systems you must provide some kind of identification before you are allowed to enter the system. We will implement this in our game by asking the player two codes: a high user code and a low user code.

In Figure III.5 the adapted syntax is shown for this new form of the "login" command.

Figure III.5

```
login IF NOT EQUAL CURLOC 11 THEN MESSAGE 1
      ELIF EQUAL LOGIN 1 THEN MESSAGE 3
      ELSE MESSAGE 6
           READI DUMMY1
           MESSAGE 7
           READI DUMMY2
           IF EQUAL CODE1 ?DUMMY1 AND EQUAL CODE2
      ?DUMMY2
           THEN SETVAR LOGIN 1
                MESSAGE 2
           ELSE MESSAGE 8
           FI
      FI
END

variables:  DUMMY1 DUMMY2 CODE1 CODE2
messages:   #6 high code number please:
            #7 low code number please:
            #8 nice try, better luck next time.
```

Explanation:

the player has to be at location 11 (where the main terminal is). If he is and did not use "login" before the ELSE part is executed, message 6 will be printed on the screen; the statement "READI DUMMY1" causes the number which is typed by the player to be stored in the variable DUMMY1. The same applies to message 7 and DUMMY2. In the next IF statement the values entered by the

player are compared with the values in the variables CODE1 and CODE2. These variables contain a combination of numbers chosen at random (how we obtain random numbers is explained in the next section). If there is a match, the "login" is accepted like we did in the previous section, otherwise the player is informed with message 8 that his guess was wrong.

The way we wrote the "login" command permits the player to try as many times as he likes. A simple modification, however, will prevent him from doing so.
We store the number of guesses he makes in a variable NLOG, and use the syntax shown in Figure III.6 instead of the line ELSE MESSAGE 8.

Figure III.6

```
ELSE IF EQUAL NLOG 3 THEN MESSAGE 9
                          QUIT
                     ELSE MESSAGE 8
                          ADD NLOG 1
        FI
```

variables: NLOG
messages: #9 You had three tries, but you blew it. Don't worry, it's only the end of the world...

Explanation:

if the variable NLOG is equal to three the player gets message 9 and the game is over, otherwise he gets message 8 and his attempt will increment the variable NLOG. Note that the combination ELSE IF can also be written as ELIF, in which case the last FI must be discarded.

III.6. The user identification codes

In the previous section we used two variables CODE1 and CODE2 which needed to contain a random combination of two numbers. We now will explain how such a combination can be set, and how the player can acquire these codes somewhere.

The basic idea is to make the player obtain these codes from persons present in the game. We choose a "system manager" and a "system engineer" for these persons. The command to use is: "ask code". There are two ways of implementing the two persons:

- by referring to them in the location description, in which case they are not real objects with properties that can be modified (static objects),
- by introducing them as objects with properties (dynamic objects).

We will implement the first alternative and then modify one of the persons, the "system manager", into a dynamic object, with which we will create another puzzle.

The word "code" has to be defined as an object, because it is used with the transitive verb "ask" (the result is shown in Figure III.7). SKIP means that no special condition is required (like for instance INVENT or LOCAL).

Figure III.7

```
ask code SKIP
    IF EQUAL CURLOC 8 THEN
       SETRAN CODE1 99
       MESSAGE 10
       PRINT CODE1
    ELIF EQUAL CURLOC 6 THEN
         SETRAN CODE2 99
         MESSAGE 11
         PRINT CODE2
    ELSE MESSAGE 12
    FI
END

transitive verbs: ask       1
objects:          a code 1 NOCLASS 255 255
messages:  #10 The low code part is:
           #11 The high code part is:
           #12 There is no one to ask a code here.
extensions to location strings:
           #8 ....The system manager is sitting in his chair.
           #6 ....A service engineer is at your service.
```

Explanation:

At location 6 or 8 the command "ask code" is applicable; at other locations message 12 is printed. The statement "SETRAN CODE1 99" picks a random value between 0 and 99 and stores it in CODE1 (likewise for CODE2). The appropriate message (low or high user code) appears on the screen and the "PRINTI" statement prints the random value. If the player uses the two values thus found in a "login" command, his login will be accepted. The object code does not need to have an object string and it does not need to be at a certain location, which is indicated by location and string number 255, representing the store room.

After all the indicated strings and words have been added and the syntax has been extended, we can again invoke the scanner and test the new "login" process.

The next step is to introduce an object "manager". He will not respond straight away to your request, because he is too sleepy. The player will first have to provide coffee for the manager, before he is given the code. The coffee must be fetched from the coffee machine, which will be situated at location 5: the canteen. The object "manager" has two object strings; the first one is used for the initial state in which he is too sleepy, the second one describes him after he received his coffee. Because "manager" has become an object now, described by an object string, the extension made to location string 8 made above has to be removed again.

objects: a manager 2 FRIEND 8 1
object strings: #1 The manager is sitting in his chair
#2 The manager is hacking away at his terminal like a madman

We use these ingredients to adapt the syntax for the command "ask". The left hand side of Figure III.8 shows the original, and the right hand side shows the new version.

Figure III.8

```
ask code SKIP                    ask code SKIP
    IF EQUAL CURLOC 8 THEN           IF EQUAL CURLOC 8 THEN
       SETRAN CODE1 99                  IF OBJSTR manager 1
       MESSAGE 11 THEN                  MESSAGE 13
       PRINT CODE1                      ELSE SETRAN CODE1 99
    ELIF etc.....                          MESSAGE 11
                                           PRINT CODE1
                                           FI
                                     ELIF etc.....
messages: #13 The manager does not react, he is too sleepy..
```

Explanation:

a test is made whether or not the object string of the "manager" has string number 1. If this is the case he is still too sleepy and message 13 is printed; of course, the code will not be shown along with it. Otherwise he gives the high code number just like in the previous version.

In the next section we will describe how a player can get coffee and how he can present the coffee to the "manager"; this will modify the description string of the "manager" and thus changes the state of the object "manager" in such a way that the man gives the code when asked.

III.7. The coffee machine

We now proceed to put a coffee machine in the canteen. We do not have to define an object "coffee machine", but instead we add the next line to the location description of location 5:

There is a coffee machine here, which requires coins.

We now need an object "coffee". This object must be movable because we want to take it to the room of the system manager. As with all coffee machines the player will have to insert a coin to get the coffee, and in our first version we will put this coin at location 2. In a next version the player must obtain this coin somewhere, which in itself is another puzzle. Both coffee and coin are aids to the player which will be indicated by the class attribute AID. We will make use of this attribute in the command "take". The location of the coffee is 255, indicating that it is still in the store room. The coffee will appear at location 5 only after insertion of the coin. Each time the player enters location 2, the string "There is a coin here" will be printed after the description of the location itself (provided the coin is still present).

objects:	coffee 3 AID 255 3 coin 4 AID 2 4	
object	#3 There is some hot coffee here #4 There is a coin here	*strings:*

To manipulate these aids we will need the commands "take" and "drop". To give the coffee to the manager we will need the command "give". Finally to put the coin in the coffee machine we will need the command "insert".

The player will now have to execute the following sequence of commands:

- go to location 2,
- take coin,
- go to location 5,
- insert coin,
- take coffee,
- go to location 8,
- give coffee,
- ask code.

In Figure III.9 the syntax description for both "take" and "drop" are shown together, because of the great similarity between them. The Figure shows that they have an opposite meaning.

Figure III.9

```
take  CLAY LOCAL IF TESTCL CLAY AID
                 THEN CHLOC CLAY 0
                      MESSAGE 14
                 ELSE MESSAGE 15
                 FI END
drop  CLAY INVENT CHLOC CLAY ?CURLOC
                  MESSAGE 14
                  FI
END
```

transitive verbs: take 2
drop 3
messages: #14 OK !
#15 It, he or she cannot be taken.

Explanation:

the verb "take" can be applied to all objects, which is indicated by the word CLAY. Everywhere in the syntax description where CLAY appears the object, which the adventure player wants to take, is substituted. The object condition LOCAL means that the object has to be present at the current location.
A test is made whether the object is an AID, and if it is its location is changed to 0, which is the inventory. Because now the object is no longer at the current location its description string will no longer be listed on entering the location. The player also gets message 14 which confirms the "take" command. If it is not an AID then message 15 appears and nothing happens. In case of "drop" the object has to be INVENT or in the inventory; if it is its location is changed to ?CURLOC or the current location. Again message 14 confirms the success of the "drop" command.

Using these commands the player can take and drop the "coin" at any location. With the coin in his possession he has to go to location 5 and enter the command "insert coin". The syntax for this command is shown in Figure III.10.

Figure III.10

```
insert coin INVENT IF NOT EQUAL CURLOC 5
                   THEN MESSAGE 16
                   ELSE CHLOC coffee 5
                        CHLOC coin 255
                   FI
END
```

transitive verb: insert
messages: #16 There is no vending machine here

Explanation:

the first condition that needs to be satisfied is the object condition INVENT, that is, the player has to carry a "coin". If he has, but if he is not at location 5, message 16 will be printed, otherwise the location of the "coffee" is changed from 255 to 5. The "coin" is moved to location 255, that is, it is removed from the game. It may be used again later on, if needed.

With these new additions we can now get coffee and take it with us to the room of the system manager. The command that completes this first complicated puzzle is: "give coffee". Its syntax is shown in Figure III.11. From now on we assume that the use of the object condition is clear and skip its explanation.

Figure III.11

```
give coffee INVENT IF NOT PRESENT manager
                    THEN MESSAGE 17
                    ELIF OBJSTR manager 2
                         THEN MESSAGE 18
                              ELSE CHOSTR manager 2
                               CHLOC coffee 255
                               ADD SCORE 5
                               MESSAGE 19
                    FI
END
```

transitive verbs: give 4
messages: #17 Only the system manager drinks coffee, and he is not here
#18 No time for coffee boy, I'm busy
#19 The manager grabs your coffee, gulps it down, and turns into a human being again

Explanation:

if the manager is not present print message 17; if the manager has already had his coffee, print message 18,otherwise:

–change the object description string of the "manager"
–remove the "coffee" from the game
–add 5 points to the score for this magnificent achievement
–print message 19.

After a succesful "give coffee" command the manager is awake (indicated by object string number 2). In this state he will answer the command "ask code".

III.8. Hiding the coin

Just walking around and picking up a coin somewhere is not particularly exiting; we can make another puzzle out of it. We will describe the puzzle first: in the hall of the centre is a receptionist; the receptionist is not at your service, because he is playing pacman. First you have to call him after which you can buy a coin from him. To buy the coin for the coffee machine you will need money. The money is outside the building at location 1; because you can go in but not out again, you must have picked up the money before you went in. Any adventurer with some experience will do this right away, because he knows that every object in the game is usually there for a purpose. If we introduce an object "money" at location 1 with class AID, the player can use the command "take" to pick it up. To use the money we will first add the receptionist and the command to attract his attention. In Figure III.12 the new command "call" is introduced.

Figure III.12

```
call receptionist LOCAL IF OBJSTR receptionist 5
                         THEN CHOSTR receptionist 6
                              ALARM alarm1 4
                         ELSE MESSAGE 20
                         FI
END
alarm1   CHOSTR receptionist 5
END

     intransitive verbs:alarm1              3
     transitive verbs:call                  5
     objects:          a receptionist       5 FRIEND          2 5
                       money                6 AID             1 7

     object strings:   #5 A receptionist is playing pacman
                       #6 The receptionist is at your service
                       #7 There is money here
     messages:         #20 I am not deaf Sir !
```

Explanation:

if you call the receptionist and he is already at your service (i.e. his object string number is not equal to 5) you will get message 20. Otherwise he will stop playing pacman and will come to the desk. However, this will only happen during a limited period. We achieve this with the statement "ALARM alarm1 4".

If the player has used 4 turns after entering the command "call

receptionist", the command "alarm1" is automatically invoked by the interpreter. This command resets the string number of the receptionist to 5, that is, the receptionist will again be playing pacman.

The above example shows the strength of the very simple action ALARM. It gives the adventure game writer the possibility to initiate actions on a predefined or randomly chosen moment, without any action by the player. An alarm is similar to a cooking alarm. It can be set as a result of some action performed by the player, or simply ar the beginning of the game. An alarm can invoke itself again or another alarm; this can cause a chain of alarms or actions. We will give more examples of the alarm mechanism in the rest of this chapter.

The next command is "buy". Its syntax is shown in Figure III.13.

Figure III.13

```
buy coin SKIP IF NOT EQUAL CURLOC 2
              THEN MESSAGE 21
                ELIF OBJSTR receptionist 5 THEN
                                          MESSAGE 22
              ELIF NOT HOLDING money THEN MESSAGE 23
             ELIF NOT OBJLOC coin 255 THEN MESSAGE 24
              ELSE CHLOC coin 0
                   MESSAGE 25
              FI
END
```

transitive verbs: buy 6 messages: #21 There is no one with coins here
#22 The receptionist is too busy playing pacman
#23 You do not possess any money
#24 You did not use the last coin I sold you
#25 You receive a coin for the vending machine

Explanation:

the syntax for "buy" presents a somewhat more complex chain of ELIF statements. If you read it sequentially, however, it will be perfectly clear what is meant. It is just a sequence of tests:

- *if you are not at location 2 you cannot buy coins,*
- *if the receptionist is playing pacman you will get message 22,*
- *if you do not have any money you will be notified of this fact,*
- *if there is already a coin in the game, a new one will be refused,*
- *finally, if none of the above conditions are true you are rewarded with a coin, that is, the coin is transferred to your inventory (the ELSE part).*

To summarise: in the previous sections we gradually built up a series of puzzles around the initial straightforward ''login'' command. If you have not yet tested the game step by step, it is now a good moment to put together all the bits and pieces in the appropriate files and to play the game in its present state. Put all the commands and objects in the appropriate sections of the dictionary; add all the messages and object strings to the string file; add the syntax constructions to the syntax file and invoke the scanner. After the removal of any typing errors a coded adventure will be produced, which can be played using the interpreter.

In the next sections we will extend the ''run'' command using the same step by step process, but as an intermezzo we will first give some help commands with which the player can examine things or request additional information.

III.9. Intermezzo: help commands

In chapter II we already presented examples of commands like "look" and "invent". In this section we show them just for completeness, without further explanation. Other possible help commands are "examine" and "help". The syntax is in shown Figure III.15.

Figure III.14

```
look    LISTLOC ?CURLOC
END
invent LISTLOC 0
END

intransitive verbs: look 4
                    invent 5
```

Figure III.15

```
help CASE CURLOC
       OF 2      : MESSAGE 26
       OF 6, 8   : MESSAGE 27
       OF 5      : MESSAGE 28
       OF 11     : MESSAGE 29
       DEFAULT : MESSAGE 30
                 ADD SCORE 2
     ESAC
     SUB SCORE 2
END
examine CLAY LOCAL IF TESTCL CLAY AID THEN MESSAGE 31
                      ELIF TESTCL CLAY FRIEND THEN
                      MESSAGE 32
                      ELSE MESSAGE 33
                      FI
END

intransitive verbs: help 6
transitive          examine 7      verbs:

messages: #26 You can buy something here, but you have to attract the
            attention of the receptionist first.
          #27 This person knows something you would like to know.
          #28 Try to get coffee out of the vending machine.
          #29 You have to login and run a program.
          #30 Sorry, no help for this location.
          #31 It is an aid, use it whenever possible.
          #32 It is a friendly person.
          #33 Nothing special about this object.
```

Explanation:

''help'' is only available at locations 2, 5, 6, 8, and 11. At other locations the default message is printed. In all cases 2 points are subtracted from the score, which is kept in the variable SCORE. Because it does not seem fair to make the player pay for help he does not get, 2 points are added again in the default case.

The command ''examine'' is applicable to all objects in the game and gives some information about these objects. A much more precise description could be given, but this is left to the adventure game writer.

III.10. Extending the 'run'command

The first extension of "run" is a simple one, and in fact not really a puzzle; we just put it in for fun. The syntax is shown in Figure III.16. Its first part is equal to the original definition in section III.4.

Figure III.16

```
run IF NOT EQUAL CURLOC 10 THEN MESSAGE 1
    ELIF NOT EQUAL LOGIN 1 THEN MESSAGE 4
    ELSE MESSAGE 34
         READI DUMMY
         IF NOT SMALLER DUMMY 4 THEN MESSAGE 35
         ELIF EQUAL DUMMY 0 THEN MESSAGE 5
                                 QUIT
         ELSE ADD DUMMY 35
              MESSAGE ?DUMMY
         FI
    FI
END

messages: #34 You can select the following possibilities:
                shutdown: 0        BASIC:   1
                FORTRAN: 2         PASCAL: 3
          Type a number to select a menu item:
          #35 Wiseguy
          #36 On a mainframe ? Tut tut
          #37 There is no time for processor consuming tasks
          #38 You have a higher education, I suppose
```

Explanation:

only the ELSE part has been modified. The player gets message 34 and the number he types is stored in the variable DUMMY. If the number is not one of the possibilities, the player is warned that trying to be smart has no effect. If he selected 0 or shutdown, the game is over, as in the original version of "run". If 1, 2 or 3 is selected, 35 is added to this value and the result is used to print the corresponding message (36, 37 or 38).

Until now "shutdown" was a program available on the computer in the game. In our next puzzle this is not the case: the player has to load it first from some load medium. Therefore we prohibit the execution of item 0 of "run" as long as the shutdown program is not loaded. We will keep the possible state of the shutdown program in the variable LOADED. To make it more difficult for the

player we will demand two different load media, both of which must be present before a load will succeed. The media are a floppy disk and a magtape, both of which will be placed somewhere in the game. The floppy disk will have to be inserted in a disk unit and the magtape must be mounted on a magtape unit. Both units will be put in the second computer room (location 10), which is not in the same room as the main terminal (location 11), in order to confuse the player.

To pick up the media the player can use the command ''take''. To mount the magtape he will have to use the command ''mount'' and to insert the floppy disk the command ''insert''. This will also be the first example of a transitive verb, which can be used in combination with more than one specific object (take and drop can be applied to all objects).

First we make the small modification to the ''run'' command: instead of

```
MESSAGE 5 QUIT
```

to indicate that the player has solved the game, we put in:

```
IF EQUAL LOADED 0 THEN
   MESSAGE 39
ELSE MESSAGE 5
     QUIT
FI
```

variables: LOADED
messages: #39 MCL – > Task not YET in system

If LOADED still has the value zero (i.e. the shutdown program has not yet been loaded), the player gets message 39 as a hint, otherwise the game is over as before. We will now provide a floppy disk and a disk unit, a magtape and a magtape unit, and the commands ''mount'' and an extension to ''insert''.

III.11. The magnetic media

The floppy disk unit and the magtape unit are implemented in a similar way as the vending machine; we add their presence to the location description. The location is the computer room (location 10):

location string:

#10 You are in a computer room
There are a disk unit and a magtape unit here.

The floppy disk is placed at location 9: the supply room, and the magtape at location 12: the little secret bunker (where else would you put important magtapes !). Both the floppy disk and the magtape can be in three states:

- in the inventory
- at a certain location
- inserted or mounted

If an object is in the inventory and we enter the command "invent", only the name of the object is listed (no description string is needed). In both other cases we need a description string for the magtape and the floppy disk: one for when they are lying around, and another one for when they are inserted or mounted. Both "insert" and "mount" will change the object string. The object definitions are shown in Figure III.17.

Figure III.17

```
objects: a floppy          6 AID  9  8
         a magtape         7 AID 12 10
object strings: #8 There is a floppy disk here
                #9 A floppy has been inserted
                #10 There is a magtape here
                #11 A magtape has been mounted
```

Figure III.18

```
mount magtape  INVENT IF NOT EQUAL CURLOC 10
                      THEN MESSAGE 40
                      ELSE CHOSTR magtape 11
                           CHLOC magtape 10
                           MESSAGE 15
                      FI
END

transitive verbs: mount 8
messages: #40 There is no magtape unit here
```

The player can pick up the magtape at location 12, take it to location 10 and enter the command "mount magtape". The syntax for "mount" is shown in Figure III.18.

Explanation:

location 10 is the only one with a magtape unit present. At that particular location the mount command results in a move of the magtape from the inventory to location 10.

To insert a floppy we do not have to add a new command, because "insert" already exists. We extend the syntax of "insert" using the NEXT construction:

NEXT <object> <statement>

This extension, as shown in Figure III.19, has to be inserted before the word END.

Explanation:

the syntax for "insert floppy" has the same structure as the syntax for "mount magtape". With this extension "insert" can be applied both to "coin" and "floppy".

Figure III.19

```
insert coin INVENT ......
                   ......
NEXT   floppy INVENT IF NOT EQUAL CURLOC 10
                     THEN MESSAGE 41
                     ELSE CHOSTR floppy 9
                          CHLOC floppy 10
                          MESSAGE 15
                     FI
END

messages: #41 Inserting floppies requires a disk unit, stupid.
```

Clever adventurers will notice that after "insert floppy" or "mount magtape" it is possible to enter the commands "take floppy" or "take magtape", both of which would leave the object strings in an incorrect state. The complete "red button" game shown in Appendix B will handle this situation correctly.

To load the program into the system we need the command "load": it is the missing link between a successful "run shutdown" command and the player's efforts in getting the floppy inserted and the magtape mounted.

III.12. Loading the shutdown program

A "load" results in a test whether or not both magnetic media are in the appropriate state, and the variable LOADED will be set accordingly, which will pave the way for the execution of shutdown. Like the "run" command we also make "load" intransitive to avoid making things too complex. The syntax is shown in Figure III.20.

Explanation:

a load is only possible at location 11 where the main terminal can be found; the second condition states that the player must be logged in. The second ELIF performs a test whether or not the tape has been mounted: if not, one of two error messages is printed, depending on the state of the floppy disk (messages 42 and 43). If the tape has been mounted, but the floppy has not been inserted, again an error message is printed (message 44). In the ELSE part everything is OK: LOADED is set to one.

Figure III.20

```
load IF NOT EQUAL CURLOC 11 THEN MESSAGE 1
     ELIF NO EQUAL LOGIN 1 THEN MESSAGE 4
     ELIF NOT OBJSTR magtape 11 THEN
          IF NOT OBJSTR floppy 9 THEN MESSAGE 42
                                 ELSE MESSAGE 43
          FI
     ELIF NOT OBJSTR floppy 9 THEN MESSAGE 44
     ELSE SETVAR LOADED 1
          ADD SCORE 5
          MESSAGE 45
     FI
END
```

intransitive verbs: load 7
messages: #42 There is no medium to load from
#43 The floppy clicks, but nothing interesting happens
#44 The tape moves, but a load fails, there is something else needed
#45 Shutdown is loaded in main memory, it is getting hot..

We add one more puzzle to complete our small game: we put a door between locations 11 and 12. The player has to open the door before he is allowed to pick up the magtape at location 12. The door is locked, however. In most adventure games locked doors can be opened using keys, but that is a bit too old fashioned for a computer centre. The door has a combination lock and the player should know the combination. He is, however, much too sleepy to remember the code. This must ring a bell, because there is a vending machine with coffee, which the player will discover sooner or later. In the next sections we will present the implementation and give the solution of this puzzle.

III.13. The door to the secret bunker

We first adapt the transition to the bunker. The player can go from location 11 to location 12 with the command "in". We will make this transition conditional. In the transition table we replace the line:

```
11 s 3 in 12 ALLMOV 10
```

by the line:

```
11 s 3 in  0 ALLMOV 10
```

The transition is now conditional (remember, it is not a transition to location 0). When you create a conditional transition you have to describe it in the syntax. The description dealing with an open or closed door between location 11 and 12 is shown in Figure III.21.

Figure III.21

```
in IF EQUAL CURLOC 11 AND OBJSTR door 12 THEN MESSAGE
                                                    45
   ELSE SETVAR CURLOC 12
   FI
END

objects: a door 8 NOCLASS 11 12
object strings: #12 The door is closed
                #13 The door is open
messages: #45 Can't you read ? THE DOOR IS CLOSED !
```

Explanation:

As in previous examples the door has two possible states: closed (default) and open. The player is only allowed to enter location 12 if the door is open.

We now have a closed door. To allow the player to open it we define the command "open", as shown in Figure III.22.

Explanation:

opening an open door does not make a lot of sense, and the player must have drunk a cup of coffee to do the job anyway. The value of the variable COFFEE will be set to one by a "drink coffee" command.

Figure III.22

```
open door LOCAL IF OBJSTR door 13 THEN MESSAGE 46
                ELIF NOT EQUAL COFFEE 1 THEN MESSAGE
                                                  47
                ELSE CHOSTR door 13
                     ADD SCORE 5
                FI
END
```

variables: COFFEE
transitive verbs: open 9
messages: #46 The door is already open
#47 The door has a combination lock. You know the code, but you are too sleepy to remember it.

III.14. Drinking coffee

We have already described how a player can get coffee. When you buy a coin to use for the vending machine, the money is moved to the store room (see section III.8–the command "buy"). If we remove the statement "CHLOC money 255" from the syntax of "buy", the money will remain in possession of the player, and it will give him the possibility to buy another coin with which he can acquire another cup of coffee. After this modification has been made, we can add the syntax for "drink" as shown in Figure III.23.

In the syntax description of "drink" again there is a nice application of the ALARM function. This time it is a more complicated one: it is an alarm that restarts itself. The syntax of this alarm is shown in Figure III.24.

Explanation:

drinking coffee more than once is not good for the player. So the first time he drinks, the coffee is removed and an alarm is set. This alarm is the side-effect caused by drinking coffee: the player will get a message after 6 turns. This is the first execution of "alarm2". Until that moment the value of TOILET is zero. The player gets message 50 which warns him that he had better pay a visit to the toilet. The toilet is at location 7. The value of the variable TOILET is incremented by 1.

Figure III.23

```
drink coffee INVENT IF EQUAL COFFEE 1 THEN MESSAGE 48
                    ELSE CHLOC coffee 255
                         ALARM alarm2 6
                         SETVAR COFFEE 1
                         MESSAGE 49
                    FI
END

intransitive verbs: alarm2 8
transitive verbs: drink 10
messages:   #48 Drinking too much coffee is not good for you
            #49 You feel much more awake now; it certainly will
            benefit you in your attempt to shut the system
```

The alarm re-executes itself, so after 6 turns "alarm2" is executed for the second time. If the player has been wise enough to go to location 7, then the variable TOILET will be set to 2 (this will be described below) and nothing awkward will happen. If he has not, however, message 51 will be printed and the game will be over.

If the player visits the toilet we have to set the value of the variable TOILET to 2 (if it had the value 1). That is the transition to the toilet must be made conditional. We change the corresponding transition in the transition table as follows:

```
old transition: 3 w 2 s 7 e 8 n 11
new transition: 3 w 2 s 0 e 8 n 11
```

and we add the syntax part for the verb "s" as it shown in Figure III.25. We will skip the explanation this time.

We have now completed all the puzzles in this small game. If you think of other ones, we urge you to try to implement them yourself. In order to gain experience with using ADL it is easier to make modifications to an existing game, rather than to create a new game from scratch.

Figure III.24

```
alarm2 CASE TOILET
       OF 0 : MESSAGE 50
              ADD TOILET 1
              ALARM alarm2 6
       OF 1 : MESSAGE 51
              QUIT
       OF 2 : SETVAR TOILET 0
       ESAC
END

variables: TOILET
intransitive verb: alarm2 9
messages:   #50 Drinking coffee has a side-effect: you have to go
            to the toilet.....
            #51 You didn't go to the toilet and it ain't much use
            walking around as a COMPUTER SPECIALIST with wet
            trousers.
```

Figure III.25

```
s    IF EQUAL TOILET 1 THEN
       ADD TOILET 1
       ADD SCORE 5
       MESSAGE 52
     FI
     SETVAR CURLOC 7
END

messages: #52 It is a relief to get rid of all the coffee
```

The next section is about tidying up some loose ends.

III.15. Completing the red button game

Several times we have encountered a statement in which the player receives points for some achievement. He also gets points for every new location he discovers. At the end of the game we can print the score and tell him the maximum amount of points he could have acquired.

We print a message which depends on the value of the score. Because the end of the game (QUIT) appears at several places in the syntax, it would seem better to replace the statement QUIT by the statement

REPROC quit CLAY.

This will cause the command "quit" to be executed instead. We will provide a syntax for the command "quit" and add the word to the dictionary. Of course the player can use the command "quit" directly, for example if he does not want to continue to play. The syntax is shown in Figure III.26. All exits from the game are now concentrated in this command; in order to examine whether the player did or did not successfully use the shutdown command, we introduce a variable SUCCESS.

Figure III.26

```
quit MESSAGE 53
     PRINT SCORE
     IF EQUAL SUCCESS 1 THEN MESSAGE 5
     ELSE IF SMEQUAL SCORE 20 THE MESSAGE 54
          ELSE MESSAGE 55
          MESSAGE 56
     FI
     QUIT
END

variables: SUCCESS
intransitive verbs: quit 10
messages:   #53 From the possible 32 points you have scored:
            #54 Your performance was a disaster,
            #55 You are near the solution,
            #56 fortunately Cruise Missiles don't go off that easily.
            Give it another try....
```

The final addition concerns the number of turns the player gets to complete his adventure. Suppose 70 turns is a reasonable number. We can set an alarm at the first move of the player, by making the first transition conditional.

The first transition was:
1 in 2
which we change to:
1 in 0

Because "in" already has a conditional transition (from 11 to 12), we have to add a test to the syntax file, to see which transition we have to perform. The adapted syntax for "in" is shown in Figure III.27.

Figure III.27

```
in IF EQUAL CURLOC 1
   THEN ALARM alarm3 10
        SETVAR CURLOC 2
   ELIF OBJSTR door 8
        THEN MESSAGE 45
        ELSE SETVAR CURLOC 12
   FI
END

intransitive verb: alarm3 11
```

When the player enters the building the alarm "alarm3" is set, which will be executed after 10 turns. The syntax for "alarm3" is shown in Figure III.28

Figure III.28

```
alarm3 IF SMALLER NTURNS 70
       THEN SETVAR DUMMY 70
            SUB DUMMY NTURNS
            MESSAGE 57
            PRINT DUMMY
            ALARM alarm3 10
       ELSE MESSAGE 58
            REPROC quit CLAY
       FI
END

messages:   #57 Number of turns left:
            #58 Sorry, you've used up too much time to be able to
            save the world.
```

Explanation:
if the player has used more than 70 turns, his time has expired and

the game is over. If not, however, he still has some time left, the number (70 - number of turns) is printed and the alarm is set again after 10 turns. This will generate a sequence of alarms at turn number 10, 20, 30, 40, 50 etc..

This completes the description of "the red button game". We hope that it provides enough material to get a basic knowledge of the Adventure Description Language. In Appendix B the complete listings of the data files are presented. It is not exactly the same game as the one shown in this chapter; the goal and the structure are the same, but its commands act somewhat differently. Therefore it should still be fun to play, even after reading this chapter. In the next sections of this chapter we will give some more examples of programming techniques to exploit the possibilities of ADL.

III.16. Random transitions and random events

Until now did not use any random actions, apart from setting the user codes. This will cause the game to behave the same way every time you play it. The more predictable a game, the easier it is to play. Random transitions and events add a whole new dimension to adventure games because you can never be sure what will happen next. The first example of a random event is a vending machine which is not guaranteed to produce coffee every time you insert a coin.

The example is rather realistic (at least it is for our university) because sometimes the machine runs out of plastic cups, sometimes it does not drop one, or sometimes the cup is not in the right position after it has dropped. In this case you either see your coffee pouring down the drain, or you burn your fingers trying to adjust the cup, which is usually rewarded with half a cup of coffee or even less.

The syntax description showing how you can write this in ADL is presented in Figure III.29.

Figure III.29

```
insert coin INVENT SETRAN DUMMY 99
                   IF SMEQUAL DUMMY 25 THEN MESSAGE
101
                   ELIF SMEQUAL DUMMY 35
                   THEN MESSAGE 102
                        CHLOC cup ?CURLOC
                   ELSE CHLOC coffee ?CURLOC
                        MESSAGE 103
                   FI
END
```

objects: cup has to have a definition
messages: # 101 Well it happens sometimes, the machine didn't eject a cup and you see your coffee pouring down the drain.
102 You burn your fingers trying to adjust the cup that didn't come down properly, but all you get is an empty cup.
103 The aroma of freshly made coffee enters your nostrils.

The second example comprises some possible random transitions. First we consider a corridor with three doors. When the command "enter" is used the player will be transferred to one of the three rooms. The part of the transition table describing this situation is shown in Figure III.30.

```
10 enter  0
11 out   10
12 out   10
13 out   10
```

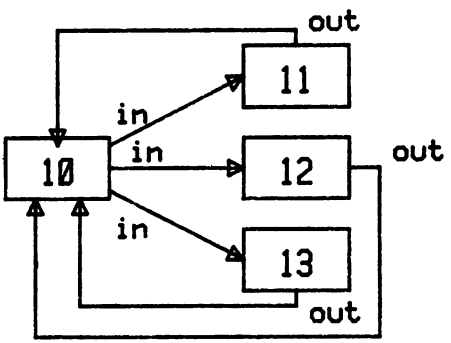


Figure III.30

The syntax that describes the random transition from location 10 to 11, 12 and 13 is shown in Figure III.31.

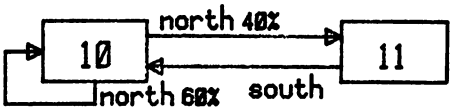
Assign a random number between 0 and 2 to the variable DUMMY, add 11 to it, resulting in one of the values 11, 12, or 13, assign this value to the current location.

Figure III.31

```
enter SETRAN DUMMY 2
      ADD    DUMMY 11
      SETVAR CURLOC ?DUMMY
END
```

Next we consider a transition with a certain probability, as shown in Figure III.32.

```
10 north  0
11 south 10
```




```
north SETRAN DUMMY 99
      IF SMALLER DUMMY 40 THEN SETVAR CURLOC 11
      FI
END
```

Figure III.32

There will be 40% chance that the transition to location 11 will occur, and 60% chance that the player stays where he is.

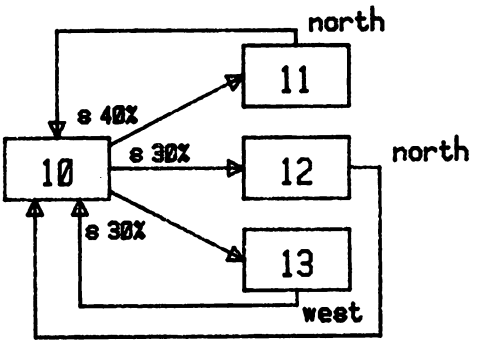
The third example is a modification of the first one. A transition occurs to one of three possible locations, each with their own probability.

From location 10 we go with 40% chance to location 11, with 30% chance to location 12, which leaves 30% chance to go to location 13. If we were to change the ELSE part to:

```
ELIF SMEQUAL DUMMY 90 THEN SETVAR CURLOC 13
```

we would go with 20% chance to location 13, and thus stay at the current location with the remaining 10% chance.

```
10 south  0
11 north 10
12 north 10
13 west  14
```




```
s SETRAN DUMMY 99
   IF SMEQUAL DUMMY 40 THEN SETVAR CURLOC 11
   ELIF SMEQUAL DUMMY 70 THEN SETVAR CURLOC 12
   ELSE SETVAR CURLOC 13
   FI
END
```

Figure III.33

Our fourth example concerns random objects. The pirate and the dwarf in the classic game of ''Colossal adventure'' are examples of random objects. The pirate appears at random and snatches treasures away from the player. The dwarf appears suddenly at the same location as the player and throws a knife at him in an attempt to kill him. The attack succeeds with a certain probability, in which case the player gets himself killed. A funeral for our unlucky adventurer is the only thing that remains to be done. How can we implement a dwarf using ADL?

We have to make some transition conditional, wihout printing any message. We only use the transition to start alarms or other actions (compare with the first "enter" in the red button game). Using alarms we can make an object like a dwarf appear suddenly, and although the dwarf was not there a moment ago, he now blocks the way.

Another technique you can use is to set an alarm which invokes itself time and time again, either after a fixed or after a random number of turns. With such an alarm you can move objects like pirates, dwarves or other enemies, but of course also friends and aids through the game, thus creating the impression that they "follow" the player. It is also possible to let them appear at a random location in the neighbourhood of the player. If it appears at the location at which the player is presently, let the object perform some fixed or random action, otherwise attract the player's attention to the fact that something is going to happen.

An example showing an unfriendly troll is presented in Figure III.34.

Figure III.34

```
alarm10 SETRAN DUMMY 4
        ADD DUMMY ?CURLOC
        SUB DUMMY 2
        CHLOC troll ?DUMMY
        IF EQUAL DUMMY ?CURLOC
        THEN MESSAGE 110
             IF RAND 50 THEN MESSAGE 111
                             REPROC quit CLAY
             ELSE CHLOC axe ?CURLOC
                  MESSAGE 112
             FI
        ELSE MESSAGE 113
        FI
        ALARM alarm1
```

objects to be defined: troll and axe
messages: # 110 a troll enters the room and throws an axe at you.
111 he hits you; oh boy you got yourself killed.
112 the axe misses you barely, hits a rock and falls to the floor.
113 somewhere a malevolent creature produces a frightening growl.

Explanation:

A random value between 0 and 4 is put in variable DUMMY and the value of the current location is added. After this 2 is

subtracted, leaving a value in the range (CURLOC −2, CURLOC +2) in DUMMY. The troll is moved to this location. If the location is the current one the troll confronts the player and he throws an axe at the poor devil. The player will be killed With a probability of 50%, and if not the axe is moved to the current location (to the floor). If location is not the current one the player will receive the message that some sinister creature is in his neighbourhood.

III.17. Recursive definitions

We already mentioned the word "recursion" when we described the syntax rules of ADL. A recursive definition provides a mechanism to program repetition. In ADL one of the actions is REPROC; this will cause a command to be executed. Using REPROC, it is also possible to invoke a command from inside its own syntax description. We call this normal recursion. Another kind of recursion is cross-recursion, in which one command invokes a second one, which in turn invokes the first one.

Repetition is a basic element in programming languages. A very important issue in connection with repetition is termination. The condition on which a repetition ends is called the "stop criterion".

An example of recursion in ADL is presented in Figure III.35.

Figure III.35

```
detonate IF NOT HOLDING dynamite
         THEN MESSAGE 150
         ELSE SETVAR DUMMY 10
              REPROC count CLAY
              MESSAGE 151
              SETVAR NOBLOCK 1
         FI
END
count    IF NOT EQUAL DUMMY 0
         THEN PRINTI DUMMY
              SUB DUMMY 1
              REPROC count CLAY
         FI
END

messages:  #150 You do not carry dynamite
           #151 You remove the wall that blocks your way.
```

Explanation:

the player can only use "detonate" if he is in the possesion of dynamite. We assign the value 10 to the variable DUMMY and (re)process the command "count". "count" tests the value of DUMMY, and if it is not equal to zero we print the value of DUMMY, subtract one from it and reexecute "count". This will

print the following sequence on the screen:

```
10
 9
 8
 .
 .
 1
```

after which the value of DUMMY is zero, and "count" will stop to reexecute itself. We return to a previous "count" etc. and finally we will return to "detonate" after which message 151 will be printed.

If recursion and repetition are not handled correctly a lot of errors can be made: resulting in incorrect actions and often creating an endless loop. In our previous example the termination condition is the value of the variable DUMMY being 0. If so "count" does not longer invoke itself and the recursion terminates. At all times be careful about the following:

- the initial value of the loop variable or the variable which indicates the number of times the recursion has to be executed: always make sure that it has a defined value. Our statement "SETVAR DUMMY 10" provides an initial value of 10,
- a step towards the final value: in our case the statement "SUB DUMMY 1" provides such a step. It is a step in the direction of 0. Change it for example to "ADD DUMMY 1" or delete it and you will have created an endless loop,
- make sure that the termination condition will be satisfied. Change the step statement in our example to SUB DUMMY 3 and you will get the sequence:

```
10
 7
 4
 1
-2
-5
etc..
```

- always make sure that the step is executed before entering the recursion. If you change the statement sequence in such a way, that the statement "REPROC count CLAY" is executed first, instead of after "SUB DUMMY 1", then you will get the sequence:

```
10
10
10
etc..
```

III.18. Putting it all together

In this chapter we have tried to give an impression of the possibilities of ADL. We discover new features, with each new game we create. We do not feel the need to extend the language, the conditions or actions. Every time we thought it impossible to program a particular construct in ADL, a closer inspection of the problem and some thinking showed that it was possible within the existing ADL, and simpler then we thought. The language is simple; it contains only a very limited set of actions and conditions, but it has proven not to be a limit on what you can program. Techniques like random choices and random behaviour, recursion and alarms are very powerful tools. The chosen model has some limitations; we did for example not provide an input interpreter to handle complex phrases. We limited ourselves to basic properties of objects. Simplicity was our device; we want adventure game writing open to a large public.

The adventure game is one of the most interesting computer games. It stimulates the creativeness and imagination of the player. A second reason for inventing a special adventure description language is to make adventure game writing available to those who have brilliant ideas, but have difficulties when it comes to writing a program to implement these ideas. Adventure games should be spread on computers in much the same way as cross word puzzles are spread on paper.

For those readers of this book who want to explore more complex adventure game models, we think that the model as presented in this book is a good starting point. It presents a structured approach; if you want to add prepositions and adjectives or if you want to process even more complex inputs, the structure can be extended very easily. Add some word categories, supply actions and conditions with more parameters and you can cope with phrases like:
open red door with key

Our model has few limitations; most computers still have too many.

CHAPTER IV

The theory behind the Adventure Description Language

In this chapter we introduce the basic theory of computing science, upon which the Adventure Description Language and Adventure Game model are based. In particular, the concepts of grammar, language, automaton, interpreter, graph and text compression are discussed.

IV.1. What is a grammar?

IV.1.1. Languages, sentences and words

''Grammars'' are related to the notion of a 'language'. Everyone has an understanding of the concept of language. Indeed most people are able to speak one or more ''natural'' languages, such as the English language. A *language* is usually a very large collection of sentences. Every *sentence* consists of a series of words in some specific order. The words which can be used in building sentences can be found in a lexicon of the language. A lexicon is a list of words together with their meaning explained in the language itself or in some other natural language.

IV.1.2. Construction of sentences

Let us consider four English words: ''is'', ''mouse'', ''small'' and ''the''. By putting these words into a specific order we can obtain, for example, the

following two *strings:*

"is mouse small the"
"the mouse is small"

It is immediately clear that there is no full freedom of choice in constructing sentences belonging to the English language. There have to be "rules" telling us which sequence of words results in a genuine English sentence.

IV.1.3. Grammars

A grammar comprises the set of rules that determine, for a given language (for example the English language), how sentences are built. The composition aspect of a particular language is called the *syntaxis*. A grammar however tells us nothing about the meaning of a particular sentence; its meaning is called the *semantics* of the sentence. In the next sentence we can see a clear difference between syntaxis and semantics. The sentence has a correct syntactic form but a rather absurd semantic interpretation:

"the mouse eats a lion"

Every grammar determines exactly one language, but it is possible to have several grammars describing the same language.

IV.1.4. An example of a language with a grammar

In many programming languages it is possible to combine commands or actions to form a new command, often called a composite command. To indicate this composition the combined actions are enclosed between the words *begin* and *end*. Let us consider a very simple example of a language called L.
This language consists of sentences containing only pairs of *begin* and *end*. Some sentences of L are:

"begin end"
"begin begin begin end end end"
"begin begin begin begin end end end end"

A closer inspection of the sentences of L reveals their *structure*. This structure can be described in a very compact way:

1) *"begin end"* is a sentence of L
2) if S is a sentence of L then *"begin* S *end"* is also a sentence of L.

S is a *nonterminal symbol*. Such a symbol is only used in describing the structure of the sentences of L. The words occurring in those sentences are called *terminal* symbols. Thus a sentence does not contain nonterminal symbols, but only terminal symbols.

When we are constructing a sentence according to the grammar rules of a

language, we will have to operate on a string consisting possibly of terminal and nonterminal symbols, for example:

"begin S *end"*

If the string does not contain nonterminal symbols it is a genuine sentence of the language, otherwise we will try to replace one (or more) nonterminal symbol(s) by a string. This operation is repeated until no more nonterminals exist in the string.
For example our string *"begin* S *end"* can be transformed by taking S and replacing it by *"begin* S *end"*, according to rule 2 above. This results in the string *"begin begin* S *end end"*. Again taking S and replacing it by *"begin end"* results in a string without any nonterminal symbols left,

"begin begin begin end end end",

and this is a sentence of L.

We can describe the grammar of L formally:

a sentence of L is a Nest, where
Nest : *begin end*
or Nest : *begin* Nest *end*

Nest is the only nonterminal symbol used here. You may read the symbol ":" as equivalent to "is of the form". With this formal grammar (consisting of two different *rules*) we have characterised the complete collection of sentences of L, although there are infinitely many sentences in that collection! This illustrates the descriptive power of grammar rules in modelling a language.

IV.1.5. An example of a grammar with its language

Let us consider a somewhat more complicated grammar; in its seven rules five different nonterminal symbols are being used, and all sentences consist of two different terminal symbols.

Figure IV.1

1.	Sentence	:	Two Sentence
2.	Sentence	:	
3.	Two	:	Ones Zero Ones Zero Ones
4.	Ones	:	One Ones
5.	Ones	:	
6.	Zero	:	"0"
7.	One	:	"1"

Note that rule number 2 states that the nonterminal Sentence may be replaced by an empty string, that is, the string "0 Sentence 11" can be transformed, using rule 2, to "011".

Starting with the nonterminal Sentence we can now construct a number of sentences according to these rules. See Example IV.2.

Figure IV.2

```
"1"
"00"
"1010"
"11010"
"111011011"
"1001010"
"1000011" etc., etc..
```

We would like to say something about the language composed of all these sentences; in other words, we would like to describe the language in words. Question: ''what have all these sentences in common?''

The answer can be obtained by looking at the grammar rules and observing the following:

1. *If a sentence has to have one or more zeros in it, we will have to use rule number 3 to introduce them*
2. *If we use rule number 3 we will always introduce two zeros!*

So this grammar describes the language consisting of all sentences containing an even number of zeros. Actually this is a rather careless statement; one must investigate whether *all* sentences with an even number of zeros can be generated, and secondly one has to examine the grammar rules to find out whether there are any restrictions with respect to the place where ones can occur and to the number of ones.

IV.1.6. Abstract definition of a grammar

Definition:

A grammar G consists of four parts:

G= (T,N,S,R) with
T= collection of terminal symbols
N= collection of nonterminal symbols
S= an element of N, which serves as a starting point for the generation of all sentences of the language
R= collection of grammar rules

As an example consider the example from IV.1.4 again.

Figure IV.3

```
G= ( T= { begin, end },
     N= { Nest },
     S= Nest,
     R= { Nest : begin end ;
          Nest : begin Nest end
        }
   )
```

IV.1.7. Generation and parsing of sentences

If we have a string consisting of a mixture of terminal and nonterminal symbols, we can try to generate a sentence from it by applying the grammar rules. Just locate a nonterminal in the string and substitute the right hand side of one of the rules for this nonterminal symbol. As an example consider again the grammar from IV.1.5 and the string "1 Zero Ones".

```
Choose Ones and rule   4  => "1 Zero One Ones"
       Zero and rule   6  => "10 One Ones"
       Ones and rule   5  => "10 One"
       One and rule    7  => "101"
```

As you can see, this string consists solely of terminal symbols.

Now we have generated a sentence of the language described by the grammar, provided the generation process started with the string containing the startsymbol of the grammar.
(Remark: the above generated sentence "101" did not start with the nonterminal Sentence as it does contain only one zero.)

Example: "Nest" => "*begin* Nest *end*"
=> "*begin begin* Nest *end end*"
=> "*begin begin begin end end end*"

If the process is carried out in a reverse order it is called *parsing.* A string of terminal symbols is transformed back to the original startsymbol. Parsing reveals the grammatical structure of a string and enables us to decide if the string is a sentence of the language or not.

Figure IV.4

```
   "01011"
=> "Ones 01011"
=> "Ones Zero 1011"
=> "Ones Zero One 011"
=> "Ones Zero One Ones 011"
=> "Ones Zero Ones 011"
=> "Ones Zero Ones Zero 11"
=> "Ones Zero Ones Zero One 1"
=> "Ones Zero Ones Zero One One"
=> "Ones Zero Ones Zero One Ones"
=> "Ones Zero Ones Zero Ones"
=> "Two" => "Two Sentence"
=> "Sentence"
```

Remark:
The parsing succeeded because we reached the start symbol, therefore we conclude that "01011" is a sentence from the language.

We obtain a clearer view of the structure of the sentence if the parsing process is shown in the form of a *parse tree* as in Figure IV.5.

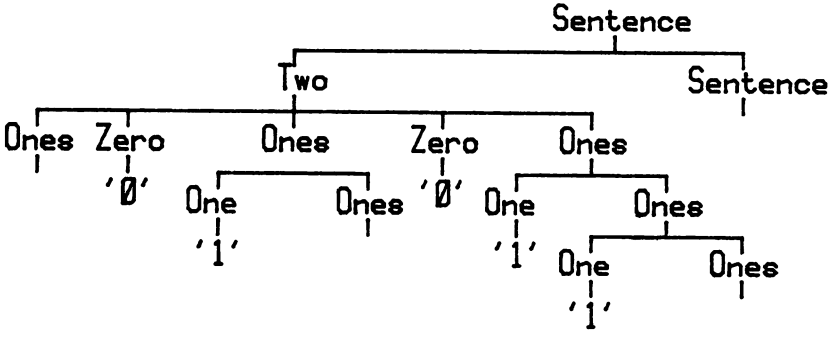


Figure IV.5

Remark:
The way the sentence is generated is also reflected in this parse tree.

IV.1.8. A hierarchy of families of grammars

In the 1950's the linguist Noam Chomsky studied grammars of all kinds of languages including natural languages. He defined a hierarchy of families of grammars based on the external form of the grammar rules. This hierarchy is called "Chomsky's hierarchy".

Class 1. The smallest and simplest family is called the set of *regular* grammars. A grammar rule of a regular grammar is of the form:

A : t B
or A : (the empty string as right hand side)
with A and B elements of N, the nonterminal symbols
and t element of T, the terminal symbols.

Parse trees for sentences generated from regular grammars have a rather simple form.

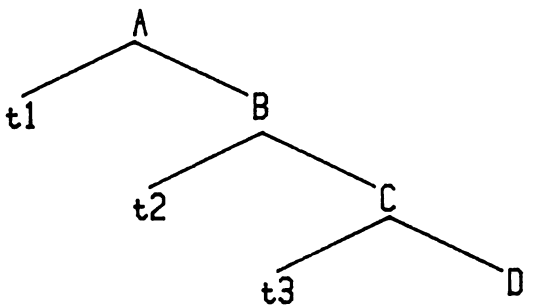


Figure IV.6

Class 2. The next family is called the set of context free grammars. This family is most commonly used in programming languages. A grammar rule of a context free grammar is of the form:

A : s

with "A" a nonterminal symbol and "s" the empty string or consisting of terminal as well as nonterminal symbols. Clearly the regular grammars are a subset of the context free grammars.

Class 3. The next family is called the set of context sensitive grammars. A grammar rule of a context sensitive grammar is of the form:

x A y : x s y

with "A" a nonterminal symbol and "x", "y" and "s" strings consisting of terminal and nonterminal symbols. An extra restriction is that the string "s" may not be the empty string.

By choosing the empty string for "x" and "y" we have the form of a context free grammar rule, so the context free grammars form a subset of the context sensitive grammars.
These rules are called "context sensitive" because the nonterminal "A" can only be replaced by the string "s" if it occurs in the context of "x" and "y", whereas in context free grammars this substitution is always allowed. So context sensitive rules can be used for more subtle substitutions.

Class 4. The largest family is called the class of *type 0* grammars. There are no restrictions in the form of a grammar rule, so it is obvious that all previous families are subset of this class of grammars.

IV.2. Representation with BNF

IV.2.1. History of BNF

We will now show a notation which can be used to represent grammars. This notation is due to John Backus and Peter Naur and is therefore called the Backus-Naur-Form (short BNF), but sometimes people refer to it as the Backus-Normal-Form, thereby not giving credit to Peter Naur. It was first used in the description of the programming language Algol60 (in 1960). Nowadays a large number of programming languages are defined in terms of BNF syntaxrules (or some dialect of BNF).

IV.2.2. Language and metalanguage symbols

If you want to describe something you will have to use symbols. In describing languages however you are confronted with the problem that languages themselves consist of symbols. So a conflict can arise between language symbols and symbols used in the description of the language. The latter are called *metalanguage* symbols and as such stand on a higher level than the symbols used in the language itself.

We have already met a metalanguage symbol, namely ":", which stands for "is of the form".

IV.2.3. BNF rules

The Backus-Naur-Form uses four metalanguage symbols, therefore they may not be part of the set of symbols of the languages described with the BNF-formalism.

Symbol	*Meaning*
::=	"is of the form", or 'may be composed of"
|	"or"
{ and }	the curly brackets denote a possible repetition of the enclosed string of symbols for zero or more times
< and >	the angular brackets enclose a symbol to indicate that it is a nonterminal symbol.

Let us now give some examples of their use.

Figure IV.7

<A> ::= <B> <C>
nonterminal A is of the form "B followed by C"

Figure IV.8

<A> ::= <B> | C
A is either a nonterminal B or a terminal C

Figure IV.9

<A> ::= <B> { +<B> }
is equivalent to
<A> ::= <B> or
<A> ::= <B> + <B> or
<A> ::= <B> + <B> + <B> or
etc. etc.

IV.2.4. An example grammar in BNF

Figure IV.10

< Sentence > ::= { < Two > < Sentence > }
< Two > ::= < Ones > 0 < Ones > 0 < Ones >
< Ones > ::= { 1 }

We have seen this grammar before in IV.1.5 in a slightly different form. In this book we use BNF to describe the Adventure Language grammar.

The next example will try to define arithmetic expressions. Some expressions we would like to describe are:

3 3+5 3+5*12 (3+5)*12 etc. etc.

It is important to note the priority rules when we calculate the value of these expressions; multiplication needs to be performed before addition, unless the addition is parenthesised. These priority rules have to be reflected in the grammar rules.

Figure IV.11

< expression > ::= <term> { +<term> | -<term> }
< term > ::= <factor> { *<factor> | /<factor> }
< factor > ::= <number> | (<expression>)
< number > ::= <digit> { <digit> }
< digit > ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Let us now parse the expression " 12−5*3 " as in Figure IV.12.

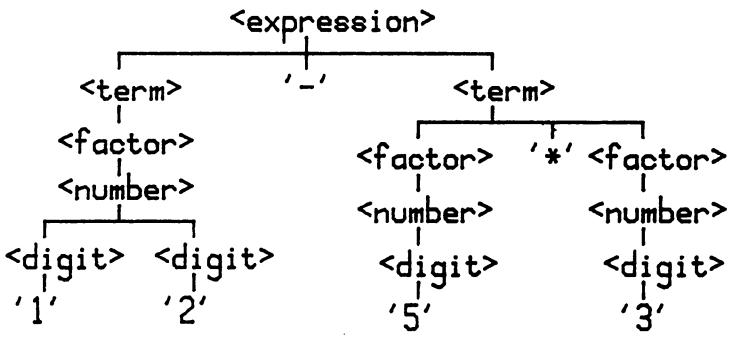


Figure IV.12

Note that the multiplication forms a subtree in the parsetree of the expression, so in order to calculate the value of the expression the value of the subtree has to be known, i.e. the multiplication must be carried out before the subtraction can be performed.
We will now give the parse tree for the expression " (12 – 5)*3 ".

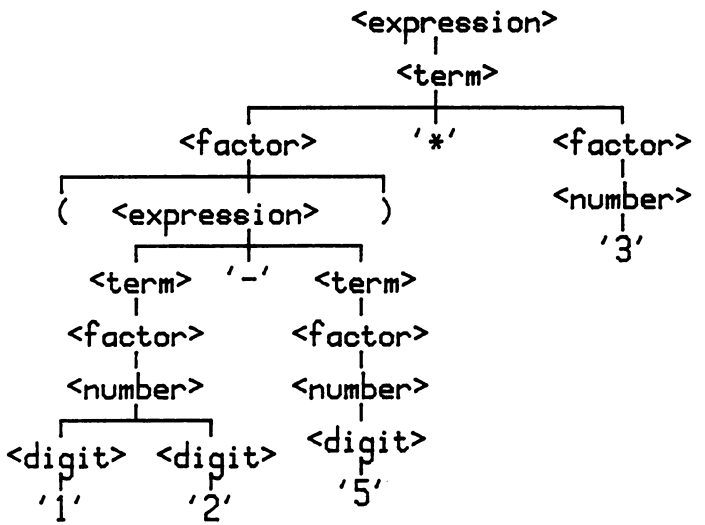


Figure IV.13

IV.3. What is an automaton?

Sometimes you encounter an apparatus which performs some function in an unknown way. Often the name "black box" is used for such a machine. The inner parts of the machine are invisible to the outside world. The only way to understand the functioning of the apparatus is to study the output in relation to its input. Some examples of black box machines:

- put a substantial amount of money in a vending machine and try to get something out of it; this should not be too difficult if the machine functions properly.
- transport a large shipment of coal to a power station and enjoy the electricity made available in your house.
- feed your chickens and see what they produce!

IV.3.1. Abstracting away from the black box machine

We will try to give a (one!) "theoretical" model for all kinds of "real" machines like the ones in the previous section. This model is called an *automaton.*

In order to describe a machine we must specify the kind of objects which are fed into the machine (the input of the machine). Usually the input objects can be divided into several classes. For example input to the vending machine, coins, can be divided into classes according to their value. This classification offers the opportunity to associate a unique symbol (or token) with each class of objects. In mathematics this would be called a code. The complete set of unique symbols is called the *input alphabet.*

Similarly the machine output objects are coded according to the so called output alphabet.

The difficult part to describe is of course the machine itself, especially the way it works. If you go to a vending machine, it is ready to accept your money; more precisely it is ready to accept the right amount of money for the selected output item. We say the machine is in its *initial state.*
After the first coin has been inserted into the machine it is no longer in its initial state. Luckily it "remembers" the fact that you inserted a coin; it actually registers the total value of the coins inserted. It does not release the selected item until at least the right amount of money has been inserted.

After the vending machine has produced its output item it returns to its initial state, ready for the next customer. So our theoretical model must be able to describe this remembering capacity of the vending machine.
An automaton is always in some *state.* This state is an element of a set of

states, i.e. the set of all possible states the automaton can be in. The automaton can change its state by accepting an input symbol (object) and ''remembering'' it in some way. In fact its ''memory'' is reflected by these states. An abstract definition of the automaton though must wait until later, however.

IV.3.2. A simple example of an automaton

We will consider an automaton describing a so called *parity* indicator. A sequence of zeros and ones, for example 100111011001010, either contains an even or an odd number of ones. If this number is even we say that the sequence has *even parity*, otherwise it has *odd parity*. So the automaton has an input consisting of a sequence of zeros and ones (in computer science known as binary-digit-pattern or bit-pattern).
The output of the automaton is transmitted to a small light-bulb: if it glows the parity is even, if it is dark the parity of the input sequence is odd. The automaton can be in two states:
1. until now an even number of ones have been seen;
2. until now an odd number of ones were put into the parity indicator.
These states will be called the ''even''-state and the ''odd''-state.
At this point we will explain the internal workings of the machine.

- if it is in the ''even''-state and the input symbol is a zero the automaton accepts the symbol and no change of state occurs. However, when the input is a one, the automaton changes to the ''odd''-state and darkens the lightbulb.
- if it is in the ''odd''-state and the input token is a one it changes state to the ''even''-state and the lightbulb will start to glow; you should be able to find out for yourself what will happen when you offer a zero as input to the machine if it is in the ''odd''-state.

IV.3.3. Abstract definition of an automaton

A *finite automaton* consists of seven parts,
FA = (I,O,Q,F,S,NEXT,OUT)
with I an alphabet of input symbols,
O an alphabet of output symbols,
Q a finite set of states,
F a subset of Q, the so called final states,
S the start state or initial state of the machine (S is an element of Q),
NEXT the next-state function,
OUT the output-function.

The NEXT & OUT functions describe the workings of the automaton, that is they specify what happens if while in state q in Q an input symbol i in I arrives.

The machine changes its state to some q' in Q and outputs an o in O. If the machine is in one of the final states this process may come to a halt.

For example the parity indicator (previous section) can be characterised as in Figure IV.14.

Figure IV.14

```
PI = ( {0,1}, {light,dark}, {Even,Odd},
       {Even,Odd}, Even,
       [NEXT(0,Even) = Even,
        NEXT(1,Even) = Odd,
        NEXT(0, Odd) = Odd,
        NEXT(1, Odd) = Even
       ],
       [ OUT(0,Even) = light,
         OUT(1,Even) = dark,
         OUT(0, Odd) = dark,
         OUT(1, Odd) = light
       ] )
```

IV.3.4. Representation of an automaton

Enumeration of the various values of the NEXT and OUT functions is not a very attractive way to represent an automaton. A more suitable method offers the *transition diagram.* Each state of the automaton is represented by a circle, and each transition from state to state is represented by an arc leading from a circle to the respective circle. The arcs are labelled with the input and/or output symbol corresponding to the NEXT and OUT functions.

For example: the parity indicator is represented by Figure IV.15.

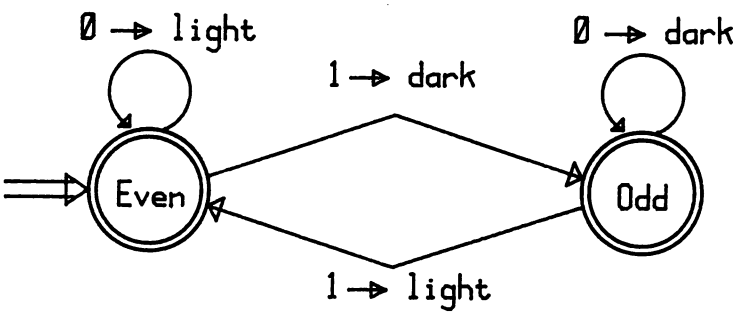


Figure IV.15

The start state is identified by a double arrow.

IV.4. Automata and Grammars

In this section we will meet some special automata, the so called recognisers and generators. The correlation between automata and grammars will be illustrated afterwards.

IV.4.1. Finite State Generator (FSG)

A finite state automaton with an empty input alphabet is called a *generator.* As an example we consider an automaton which generates all strings consisting of ''a'' and ''b'' symbols with an even number of ''b'' symbols.

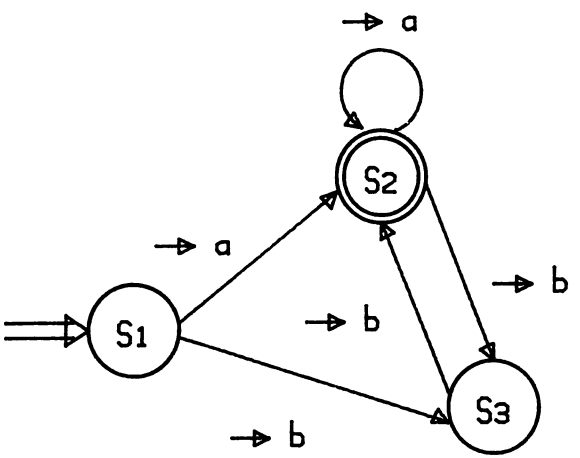


Figure IV.16

The first thing to notice is the double circle of state number S2. This marks state S2 as a *final state*, a state in which the automaton may halt (if the automaton had no final states the string generation process would have to continue forever!). The second thing to notice is the labelling of the arcs. No input symbols are specified because the automaton ignores input: it only produces output.

Let us examine another example.

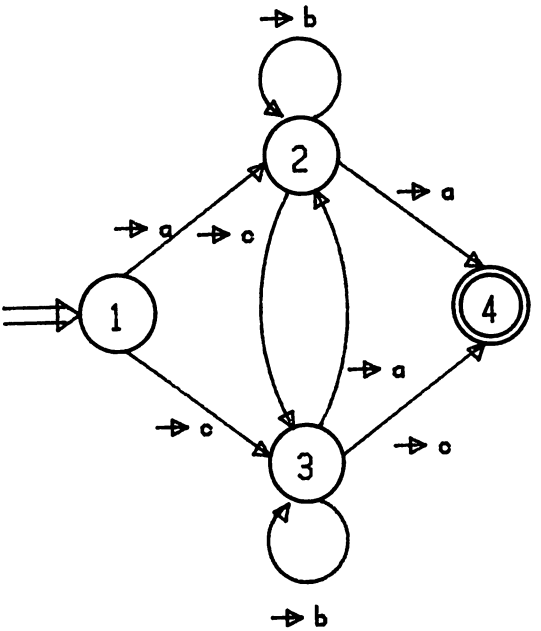


Figure IV.17

Although this diagram exhibits a nice symmetry and the automaton has only four different states, it is rather difficult to capture the language produced by this FSG. We could try to use BNF with nonterminal symbols as in Grammar IV.18.

Figure IV.18

```
< H1 > ::= a |  c {b} a
< H2 > ::= c |  a {b} c
< R1 > ::= {b} c {b} a
< R2 > ::= {b} a {b} c
< L1 > ::= < H1 > {< R1 >} {b} a
< L2 > ::= < H2 > {< R2 >} {b} c
and
< Language > ::= < L1 > | < L2 >
```

Clearly the BNF formalism is not very suitable for describing such languages. Luckily there is another notation, the *regular expression.* It uses as metasymbols

(and) for grouping
+ for choice (compare with the | from BNF)
* for zero or more iterations
(compare with the curly brackets from BNF)

juxtapositioning to indicate a concatenation and it can be easily obtained from the BNF form.

The aforementioned language can be written down as a regular expression in the following way:

Language = (a+cb*a)(b*cb*a)*b*a + (c+ab*c)(b*ab*c)*b*c

So a regular expression denotes a set of strings which together form the language.

Definition:
A Regular Expression (RE) has one of the following forms:
- A symbol, such as a,b,c,0,1...
- the empty string
- (R1)* only if R1 itself is a regular expression
- (R1+R2) only if R1 and R2 are regular expressions
- (R1R2) only if R1 and R2 are regular expressions

Note that this definition is rather strange; it *uses* the same concept it is meant to *define*. The definition indicates how regular expressions can be build from simpler ones. Let us give some simple examples to get the flavour of these regular expressions.

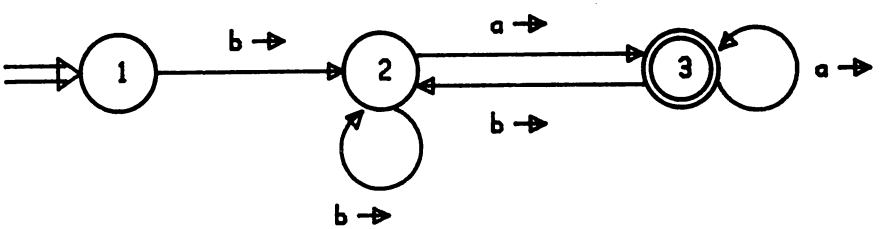


Figure IV.19

Corresponding language: b(b+aa*b)*aa*
or equivalently: bb*aa*(bb*aa*)*

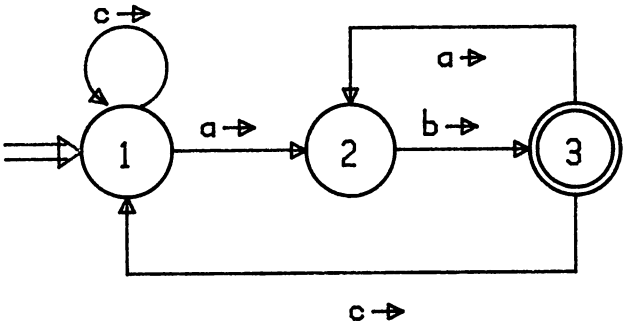


Figure IV.20

The corresponding language: $(c+a(ba)^*bc)^*a(ba)^*b$ (1) (2)

Explanation of the RE:
the first part (1) leads us from state 1 to state 1,
the second part (2) leads to the final state nr. 3.
Example IV.20 is again an example of a complicated language obtained from a simple automaton.

IV.4.2. Finite State Recogniser (FSR)

The counterpart of a generator is an automaton with an empty output alphabet, i.e. this kind of automaton produces no output at all. It just *recognises* the input string. As an example the transition diagram of a recogniser which recognises all input strings with at most three zeros and an arbitrary number of ones is shown.

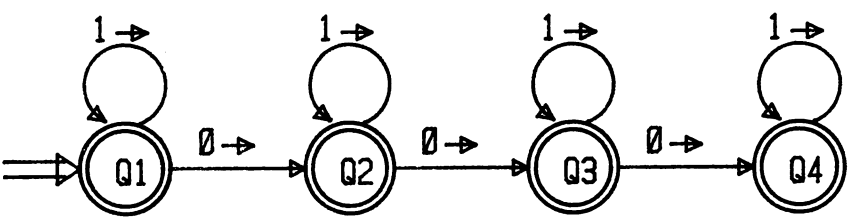


Figure IV.21

Note that the arcs are labelled only with input symbols and that no final states are shown, because the automaton is finished when the input is exhausted (completely processed).

Remark:
Not always all possible state transitions are shown. For example in the above diagram in state Q4 there is no transition for possible input symbol "0". Indeed the next state would have to be something like an error state. The machine reports an error and halts, because the input string did not conform to the form postulated above (at most three zeros in the string).

As an illustration of the close relationship between grammars and finite state recognisers the following grammar can be made which is equivalent to the automaton above.

Grammar IV.22

```
<Q1> ::= 1 <Q1> | 0 <Q2> | <empty>
<Q2> ::= 1 <Q2> | 0 <Q3> | <empty>
<Q3> ::= 1 <Q3> | 0 <Q4> | <empty>
<Q4> ::= 1 <Q4> | <empty>
<empty> ::=
```

Note the empty string alternatives for each of the final states. Another example of this relationship will be given at the end of IV.4.4.

IV.4.3. Nondeterministic machines

Apart from finite state machines there is a second group of machines called the *nondeterministic* finite state machines (NDFSM). These machines exhibit a somewhat erratic behaviour, as will be explained with the following machine. (see Figure IV.23)

Note that on input "0" the machine can choose between a transition to state 1 or to state 2; we cannot say definitely which one will be chosen. FSM's have no such choice: the NEXT-function tells us, given the current state and input symbol, which state will be the next state.
After this nondeterministic transition to either state 1 or state 2 the machine accepts the next input symbol. At this moment something strange happens: if the input symbol is a zero the machine ends in state 3, otherwise the input symbol is a one and the machine ends in state 4! This NDFSM always makes *the right choice* whenever a transition must be made. It seems as if a little devil inside the machine can look ahead in time and foresee the rest of the input. So NDFSM's have a *set* of possible next states to go to from the current state and input symbol.

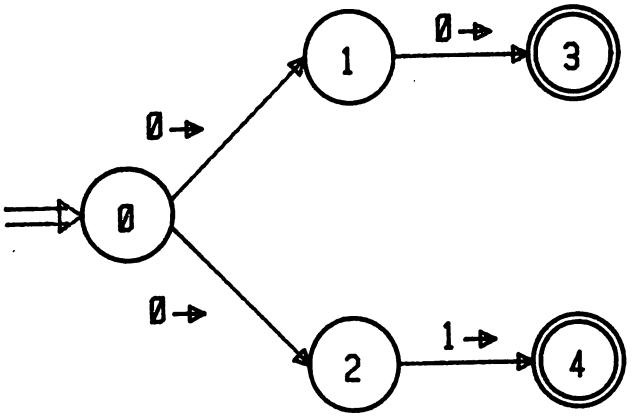


Figure IV.23

Let us consider some more NDFSM's.

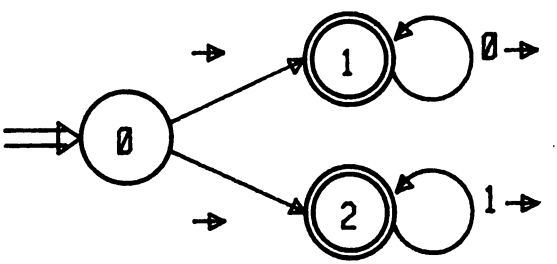


Figure IV.24

Language: (0* + 1*)

Note the transitions from state 0: no symbol is read from input. This is called an *epsilon-transition* (the empty string is recognised). These empty transitions allow us to build large machines from smaller ones by glueing them together with epsilon labelled arcs. Of course we could also build an FSM which recognises the same language. (see Figure IV.25)

Let us consider another machine with its corresponding language.

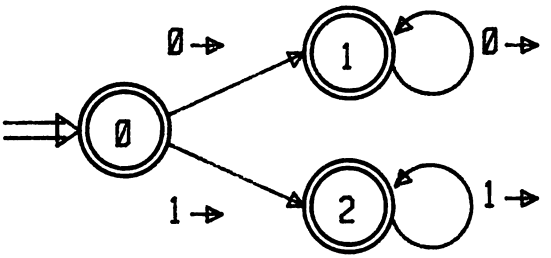


Figure IV.25

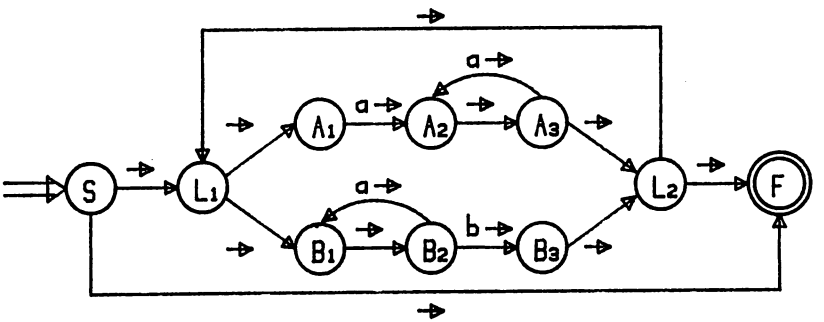


Figure IV.26

Language: (a(a)* + (a)*b)*

Note how an (R)* construct is translated into a corresponding machine component with the help of empty transitions. Although the introduction of nondeterminism seems to add something to the "power" of automata a theorem can be formulated and proved, which says:
"For every nondeterministic finite state recogniser (NDFSR) there exists an *equivalent* deterministic finite state recogniser (FSR)."

Another theorem states that:
"All NDFSR's, FSR's and regular expressions are equivalent in the sense that they describe the same class of languages, the regular languages (see IV.1.8)."

IV.4.4. Converting an automaton into BASIC

Let us now consider an automaton which recognises sequences of numbers with an odd number of zeros and an arbitrary number of ones. Let us define that the end of the string is encountered when the input token is any digit, not equal to one or zero.

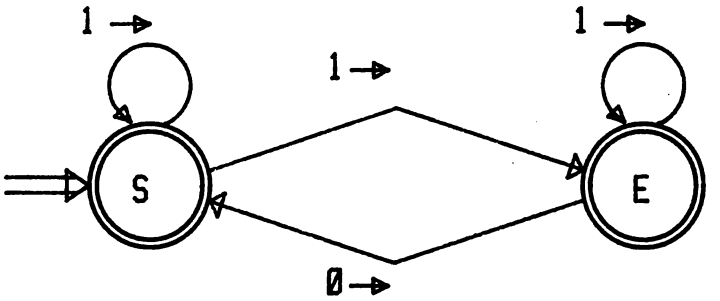


Figure IV.27

We must introduce a variable STATE to record the value of the current state, and also a variable to contain the current symbol.

```
100 REM 0 STANDS FOR S, 1 FOR E
20  STATE=0
30  REPEAT
40      INPUT I
50      IF I=0 THEN STATE=1-STATE
60  UNTIL NOT ( I=0 OR I=1 )
70  IF STATE=1 THEN PRINT"RIGHT" ELSE
                                    PRINT"WRONG"
80  END
```

Of course the automaton in this example is very simple. Let us therefore consider a somewhat more complex automaton. If the given input of the BASIC-program would lead the automaton to a final state the program prints the string "OKE", otherwise it will restart and keep silent. (see Figure IV.28)

```
10  STATE=1
20  REPEAT
30      INPUT N
40      ON STATE GOSUB 80,120,160,210
50  UNTIL NOT (N=0 OR N=1)
60  END
```

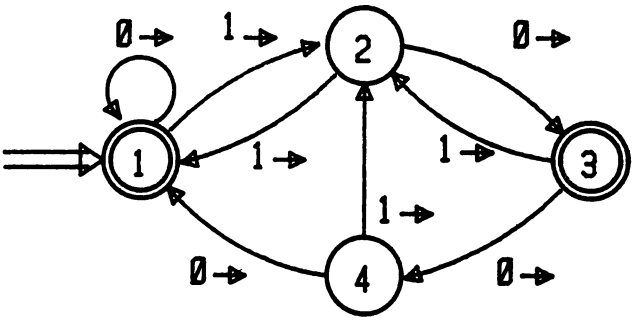


Figure IV.28

```
70  REM STATE 1
80      IF N=0 THEN GOTO 130
90      IF N=1 THEN STATE=2 ELSE PRINT"OKE"
100 RETURN
110 REM STATE 2
120     IF N=0 THEN STATE=3
130     IF N=1 THEN STATE=1
140     IF NOT (N=0 OR N=1) THEN STATE=1
150 RETURN
160 REM STATE 3
170     IF NOT (N=0) THEN GOTO 210
180     STATE=4 : ENDPROC
190     IF N=1 THEN STATE=2 ELSE PRINT"OKE"
200 RETURN
210 REM STATE 4
220     IF N=0 THEN STATE=1
230     IF N=1 THEN STATE=2
240     IF NOT (N=0 OR N=1) THEN STATE=1
250 RETURN
```

From the automaton above we can derive the Figure IV.29 as another illustration of the close relationship between grammars and FSR's.

Clearly these grammar rules obey the constraints for a regular grammar as formulated in section IV.1.8; hence the resulting language is a regular one.

Let A,B,C and D be nonterminals representing the states 1,2,3 and 4 respectively.

```
<A> ::= 0 <A> | 1 <B> | <empty>
<B> ::= 0 <C> | 1 <A>
<C> ::= 0 <B> | 1 <D> | <empty>
<D> ::= 0 <B> | 1 <A>
<empty> ::=
```

Figure IV.29

IV.5. The Adventure Description Language revisited

IV.5.1. ADL is a context free language

The grammar rules of ADL do not conform to the constraints for a regular grammar. However they do conform to the constraints for a context free grammar. Hence ADL is a context free language like most other programming languages.

As we have seen in previous paragraphs the regular grammars denote regular languages which on their part are equivalent with finite state automata. Therefore there is no finite state recogniser that accepts strings (programs) written in ADL. We can easily demonstrate this with the choice-command of ADL, the IFstatement. This feature is grammatically described by IV.30.

Figure IV.30

```
<IFstat>   ::= IF   <condition> THEN <statements>
               {ELIF <condition> THEN <statements>}
                 <elsepart>
                 FI
<elsepart> ::= <empty> | ELSE <statements>
<empty>    ::=
```

Note that in a choice-command other statements occur in the THEN-parts or ELSE-part. So one of these statements could again be a choice-command! Omitting a lot of details we will concentrate on the following grammar rule.

```
<pair> ::= ] <pair> [ | ][
```

Figure IV.31

Note that the second alternative is necessary; without it it is not possible to generate a string consisting solely of terminal symbols. Examples of strings generated by these rules:

```
[], [[]], [[[]]], ....
```

Let us assume for a moment that we could build a finite state recogniser for these bracket strings. First the left brackets are recognised and their number

is remembered, then the right brackets are matched against the left brackets. The number of left and right brackets must be equal! The only way an FSR can remember the number of left brackets seen is by going to a specific state for every number, i.e. a state to remember the fact that 2 brackets have been seen, a state for 3 brackets, one for 4 brackets, etc. etc.. The problem is that the recognised number of states cannot be specified in advance. We cannot impose an upper limit on this number of bracket pairs. Conclusion: There does not exist an automaton to recognise our ADL; therefore the ADL is not a regular language.

IV.5.2. Checking sentences against errors

A program written by someone in some particular programming language has to be checked to see if it does not contain any programming errors. We can make a distinction between several types of errors.

1) typing errors, for example 'IG'instead of 'IF'
2) syntactically corrupt constructions, for example
   IF <condition> ELSE <statements> FI
3) use of undeclared identifiers
4) interpretation (or execution) of the program leads to unex pected and unwanted behaviour; this category is often called ''programming mistake'' or ''algorithmic flaw''.

A distinction must be made between errors and symptoms. The error-checker must attempt to diagnose the program at hand on account of mere symptoms, somewhat like a doctor diagnoses a patient. As an example of this error/symptom distinction consider an undeclared or unknown identifier. It may mean a type 3 error *or* a simple misspelling of a correctly declared identifier. To pinpoint the type of error is often an impossible task, so the only sensible thing to do is to signal the symptom, guess at the right interpretation and report this to the programmer.

The class of detectable errors is largely determined by the design of the programming language. If an error is detected and reported the checking should continue, that is the error-checker has to recover from the detected error situation in such a way that the error checking continues from a certain point in the text as if no erroneous situation ever occurred.

As an example consider the ADL construct known as IFstatement. A type 2 error (syntactic error) in an IFconstruction may lead to a skip to the corresponding ''FI'' symbol, from which point normal checking can be resumed. Bracket constructs like parentheses, IF-FI, CASE-ESAC and others make this type of recovery easy. Another advantage of these pairs is the fact that a programmer may write his programs nicely indented. In general this adds considerably to the readability of programs.

Figure IV.32

```
IF <condition 1> THEN
   IF <condition 2> THEN <action 1>
   ELSE <action 2>
   FI
ELIF <condition 3> THEN <action 3>
FI
```

From Figure IV.32 anyone can see quickly the hierarchical structure of this statement.

IV.5.3. The ADL scanner and interpreter

The error-checker for programs written in the ADL is the *scanner.* Apart from reporting erroneous syntactical structures and undefined identifiers, this program also checks whether defined persons or objects have a corresponding description string, whether locations in the game can be reached at all, and whether verbs are used in the same way as they were defined (with or without an accompanying object), etc. etc..

If the game has been programmed correctly the scanner generates tables and interpretable code for the ADL *interpreter*. This means that the interpreter does not need to check its input against errors, simplifying the process of interpretation considerably ! Of course all this is described in much greater detail in section II.9. We will however mention the fact that the interpreter itself is purely a finite state automaton with a finite set of internal states, acting on its input and the current state alone. The input symbols are the ADL keywords and the user defined words (objects, persons, verbs).

IV.6. Some theory about graphs

In this section we discuss some theoretical concepts of graphs and try to give clues about the relation between graphs and Adventure Games.

IV.6.1. Introduction

Let us examine the common characteristics of various systems:

- a roadmap with cities scattered around and connected by roads
- an electrical network with interconnected (wired) components
- a state transition diagram of a finite state automaton
- a database with datarecords linked by all kinds of binary relations

These systems have in common that they can be modelled by a set of *nodes* and *interconnections* between these nodes. Graphically we can depict them as labelled inkspots connected with line segments or arrows. If we abstract away specific details we have the notion of a *graph*.

IV.6.2. Definition of a graph

A *graph* G consists of a pair of sets (V,E), where V is a set of *vertices* (or nodes) and E is a set of pairs of nodes in V, i.e. if v_1 in V and v_2 in V and (v_1,v_2) is a (pair−) element of E, then there exists an *edge* (or interconnection) between the nodes v_1 and v_2.

Often we can assign a specific direction to each edge in E. For example in a state transition diagram of a deterministic finite state automaton a state transition is an arc going from the current state (node) to the new state. Another example: if we keep potentials constant in an electrical network then the current will flow only from a node with high to a node with low potential.

Graphs with this direction property are called *directed* graphs or *digraphs* for short. In the definition of a digraph the edges are *ordered* pairs of vertices, i.e. if (v_1,v_2) in E we say the edge goes from v_1 to v_2. Often the edges of the digraph are called *arcs*. If (v_1,v_2) in E then v_1 is called the tail and v_2 the head of the arc and it is drawn in IV.33.

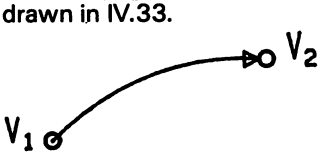


Figure IV.33

(Note that the ''arrowhead'' is at the head of the arc.)

N.B.: Note the difference in meaning between the notations of an edge for a directed and an undirected graph: in an undirected graph (1st definition) the edge (v_1,v_2) can also be written (v_2,v_1) because there is no sense of direction.

Relations can be depicted with digraphs as in the case of the database system mentioned above. If the relation is symmetrical an unordered graph can be used. Adventure Games contain a directed graph, namely the transition table. In the case of a road map all connections between cities are two-way roads, so that an undirected graph will suffice, but in the case of a map of a city there will be one-way as well as two-way roads, hence a digraph must be used to represent it.

IV.6.3. Walking around

The fun of graphs is that you can take a walk from one node to another. A *path* in a digraph is a list of vertices [v_1 , v_2 ,..., v_n] such that (v_1,v_2) in E, $(v_2,v938)$ in E, ... , (v9n-18,v_n) in E. So on a path [v_1 , v_2 ,..., v_n] there are n-1 arcs between the adjacent pairs in the list. The number of arcs is called the *length* of the path.

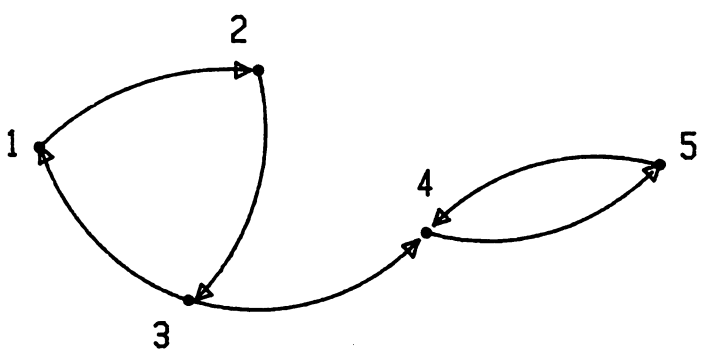


Figure IV.34

[1,2,3,4,5] and
[1,2,3,1,2] are paths

[1,2,5] is not a path

One of the problems in graph theory is the so called shortest path-problem. Given a graph G with vertices v and w in V "is there a path from v to w?" If the answer is yes we are most likely interested in the shortest path from v to w, so "give the shortest path from v to w !". If we just want to know if there is a path between any two vertices in V we are confronted with the *transitive closure* problem. It was S.Warshall who invented a famous algorithm to solve this particular problem in 1962 and in honour of that it has been named after him.

IV.6.3.1. The Warshall algorithm

Those amongst our readers, who are unfamiliar with Pascal-like programming languages, may want to skip this sub-subsection as it is not necessary for understanding the rest of this section.

We will give the algorithm of Warshall which calculates the transitive closure of a graph (or a relation, representable by a graph).

Figure IV.35

```
{   the 2-dimensional array E contains boolean values indicating
    whether or not there exists an edge from the first index (node)
    to the second; after the algorithmic evaluation has finished P
    will contain as information boolean values indicating 'if there
    exists a path between first and second index value'.
}
    { make P(aths) initially a copy of E(dges) }
    P := E;
    { n = number of nodes in the graph }
    FOR i := 1 TO n DO
        FOR j := 1 TO n DO
            FOR k := 1 TO n DO
                IF P[j,k] = false
                THEN P[j,k] := P[j,i] AND P[i,k]
    { this assignment in the THEN part states that
        if 'there is a path from j to i'
       and 'there is a path from i to k'
      then 'there is a path from j to k (over i)'
}
```

IV.6.3.2. Hamiltonian paths

Another famous problem in graph theory is the following one.

We are given a map of a part of the United States of America showing some of the larger cities and interconnecting roads. Each interconnection is labelled with the travelling distance. A travelling salesman must visit every city. Can we tell him how he must travel in such a way that the total travelling distance is minimised without visiting a city more than once, just by inspecting the graph (map)?

In mathematical terms this problem can be formulated thus: 'Let in a graph G a *Hamiltonian path* be a path $[v_1 , v_2 ,..., v_n]$ where n is the number of nodes in V, each v9i8 in V occurs exactly once on the path, then the problem is to determine:

1) whether there exists a Hamiltonian path in the given graph G
2) to determine the one with the shortest total travelling distance.

Clearly there are many graphs without even one Hamiltonian path, see for example figure IV.36.

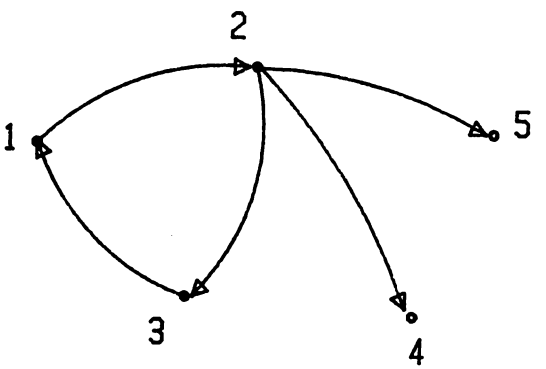


Figure IV.36

There is no Hamiltonian path
in this graph

In contrast consider the next figure carefully.

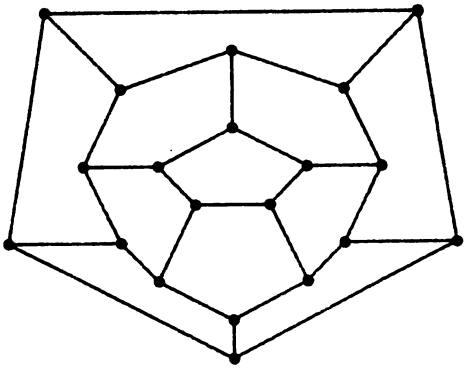


Figure IV.37

This figure contains 12 planes, the 12th being at the outside of the rim. Every vertex has 3 edges connecting it to its 3 neighbouring vertices. It contains a Hamiltonian path. Start in a randomly picked vertex. Let L designate "turn Left" and R "turn Right" then the path can be described by the formula "LLLRRRLRLRLLLRRRLRLR".

IV.6.4. Graphs divided into subclasses

As we have seen the graphs can be divided into directed and undirected graphs. According to other properties we can subdivide these two groups still further.

IV.6.4.1. Completeness

A graph G is called *complete* if every pair of vertices is in E, that is, every single vertex in V has an edge to every other vertex in V. As a consequence, if V contains n vertices there will be n*(n-1) edges in E if G is a directed complete graph; if G is undirected and complete E contains only n*(n-1)/2 edges.

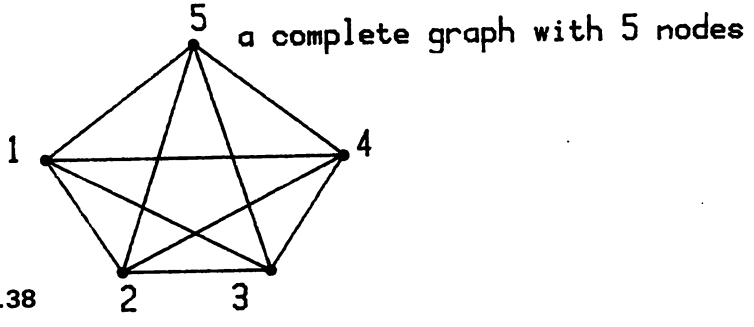


Figure IV.38

IV.6.4.2. Connectedness

A graph G is called *connected* if there exists some path between every pair of vertices in V. Clearly a complete graph is connected too, but usually a connected graph is not complete. (see Figure IV.39)

If a graph is not connected we can still subdivide the nodes and edges into connected *subgraphs.* If we translate this situation to Adventure Games, a connected graph enables the player to visit every location in the game; if the transition table does not represent a connected graph then some parts of the game cannot be reached with normal transitions. Of course some "magic" may enable the player to cross a frightening deep abyss or climb un unclimbable wall.

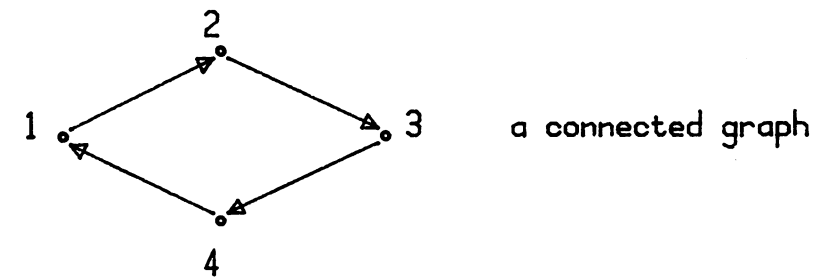


Figure IV.39

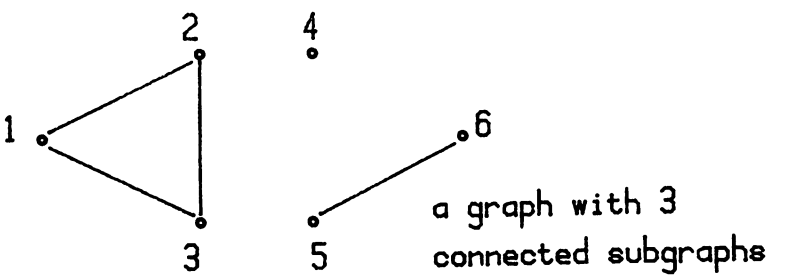


Figure IV.40

IV.6.4.3. Cyclicity

A graph G is called *cyclic* if it contains at least one cycle. A *cycle* is a path of length > = 1, which begins and ends at the same vertex. Graphs without cycles are sometimes called *acyclic* graphs.

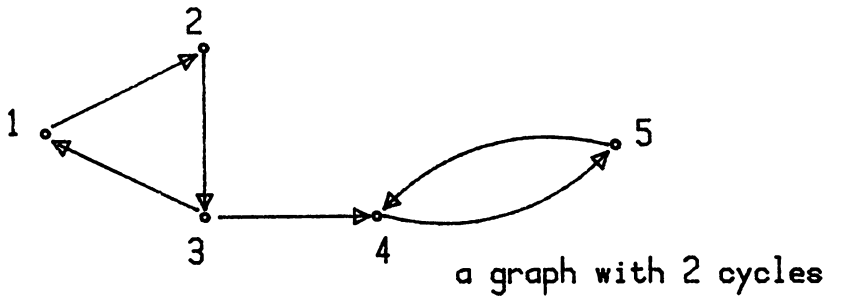


Figure IV.41

A cycle in the transition table enables the Adventurer to start a walk from a specific location and return many turns later to the same location without retracing his own steps.

IV.6.4.4. Trees

A connected graph without cycles is often called a *tree*. Trees are usually undirected, as in example IV.42.

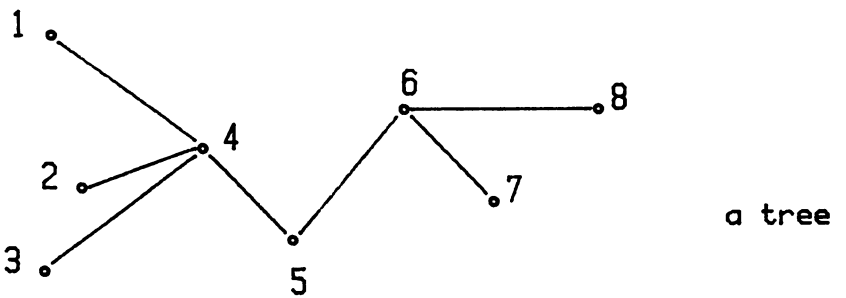


Figure IV.42

Theorem:
1. if a tree contains n vertices, it has n-1 edges
2. if we add one edge anywhere in a tree, the resulting graph will be cyclic.

It is common practice to select one particular node in the tree and call this node the *root* of the tree: roots are displayed at the top of a picture.

The other vertices are seen as children from the root, being hierarchically lower in ordering than the root. For example let us display the previous example tree with node number 5 as root.

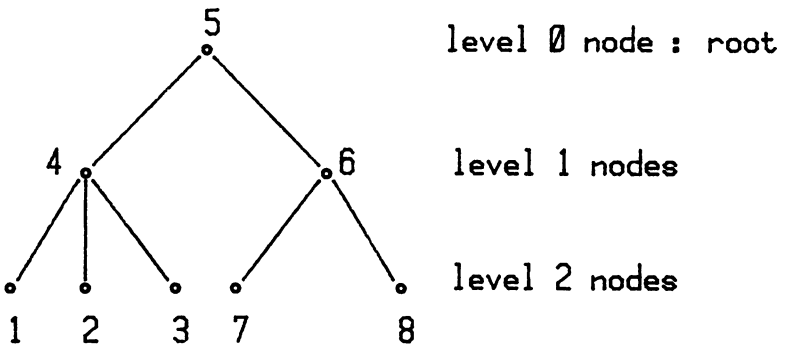


Figure IV.43

IV.6.4.5. Planar graphs

A graph G is called *planar* if its vertices V and edges E can be drawn on a two-dimensional surface, in such a way that no two edges cross each other (when drawing a picture on a sheet of paper). The graph in Figure IV.40 is an example of a planar graph, whereas the one from Figure IV.39 is not, since two edges cross in the centre of the picture.
In fact the largest *complete planar* graph is the one with only four vertices.

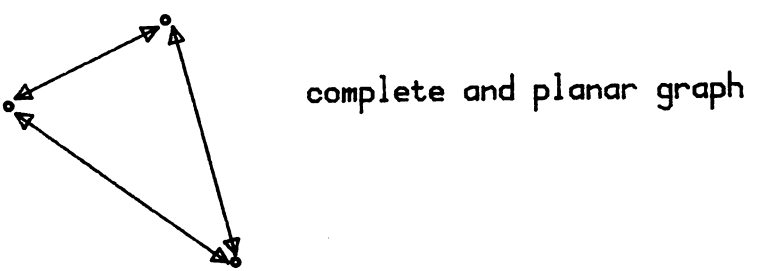


Figure IV.44

It is common in Adventure Games that nearly all of the transition table can be represented by a planar graph. This allows players starting a new adventure to draw a reasonable map, whenever they probe in various directions. There can still be a nonplanar part in the transition table, most often representing some kind of difficult maze.

IV.7. Text compression

IV.7.1. Introduction

One subject is of great importance to an adventure system on a machine with small memory, and that subject is text compression. The adventure program needs to manipulate a large amount of text, and all of this text needs to be present in the Beeb's rather limited memory. The usual way to store text in memory is to use the ASCII code, i.e. each character is encoded as one byte. This, however, is wasteful: we will be using only the "visible" characters whose ASCII codes don't exceed 127, and these characters could be coded in seven bits instead, a saving of 12.5%. It is, however, possible to do better than that. For example the total string length of the "Colossal Adventure" game currently on the market would exceed 40K if encoded in ASCII; it should be obvious that a better compression method has been used in this case.

IV.7.2. Huffman Code

One of these better compression methods is called the Huffman Code, (after its inventor) and was used to minimise the length of radio transmissions in the late 1940's. The Huffman Code is based on the fact that in an English (or in any language, for that matter) text some characters occur more frequently than others; the "e" is likely to be the most frequent character in an English text. By giving the most frequent characters a short code, say two or three bits, and the infrequent characters a longer code, we can save large amounts of memory. For example, if 50% of the text consists of the letter "e', reducing its code from eight to two bits would save 6/8 of half the size, i.e. 37.5% (and we would of course also be able to save on the size of the other half).

There are some disadvantages to using a variable length code such as the Huffman Code:

- memory needs to be manipulated at the bit level, and this is best done in machine code (BASIC would be too slow); however, programming in machine code is rather difficult;
- when decoding text, a table of all character codes is needed, and this table must be present somewhere in memory. Storing this table thus costs us some memory, reducing the gains from the Huffman Code;
- special care needs to be taken when choosing the codes. For example, it is clear that if we give "e" the code "11" and "q" the code "111111", we won't be able to tell after the first two ones whether to print an "e", or to

continue and (possibly) print a "q". Thus, the code for any character must not be the start of the code for any other character.

To summarise, all adventure text will be subject to the following:

- the frequency of all characters in the text is determined;
- for each of these characters a *Huffman Code* is determined, i.e. the most frequent characters get the shortest code;
- the original text is *encoded* using this Huffman Code.

Experiments have shown that, using the Huffman Code, English texts can be compressed to between 50% and 60% of the original size; the fact that it is also relatively easy to program has made us decide to use it for our adventure system.

IV.7.2.1. How does it work?

We shall now explain how the Huffman Code is assigned. As a simple example, let us consider a text from which the frequency Figure IV.45 is produced.

char	freq
a	6
e	8
i	5
o	3
u	1

Figure IV.45

In order to determine the Huffman Code for these characters, they are stored in a tree. (see Figure IV.46) The circular nodes in Figure IV.46 are called *internal nodes,* because there are branches emanating from them. The node at the top, marked in4 (Internal Node 4) is called the root. The square nodes are called *leaf nodes* because the tree does not branch further from them.

We will shortly explain how this tree was built from the frequency table, but first we note the following:

- all leaf nodes are marked with characters from the table, together with their frequency;
- all internal nodes are marked with the sum of all character frequencies in the tree below them; in particular, the root node is marked with the total number of characters in the text;
- there are four internal nodes, one less than the number of leaf nodes;

– the characters with the lowest frequencies are furthest away from the root.

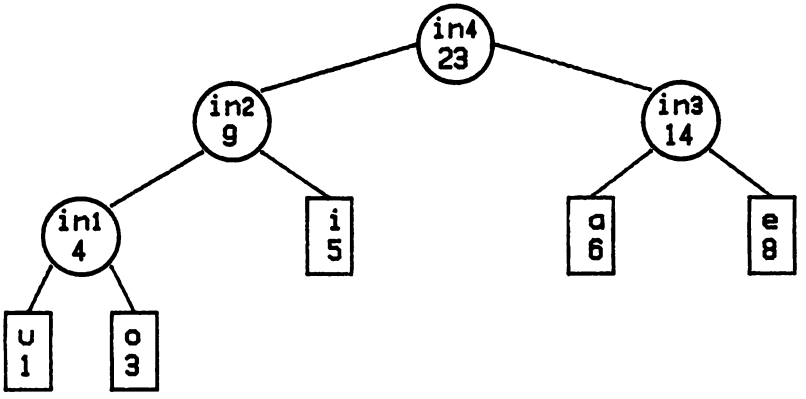


Figure IV.46

In order to extract the Huffman Code for each character from this tree, we mark each left branch with a "0" and each right branch with a "1"; the Huffman Code for each character is obtained by following the path from the root to the leaf node containing the character, and writing down the numbers encountered on the way. So the Huffman Codes in our example can be summarised in Figure IV.47.

Note that none of the shorter codes is the start of one of the longer codes; this could only happen if some leaf node was an internal node, but this is not possible by definition. Because the leaf nodes with the lowest frequencies are furthest from the root the longest Huffman Codes are assigned to them. The following amount of compression has been achieved:

char	Huffman code	ASCII code
u	000	01110101
o	001	01101111
i	01	01101001
a	10	01100001
e	11	01100101

Figure IV.47

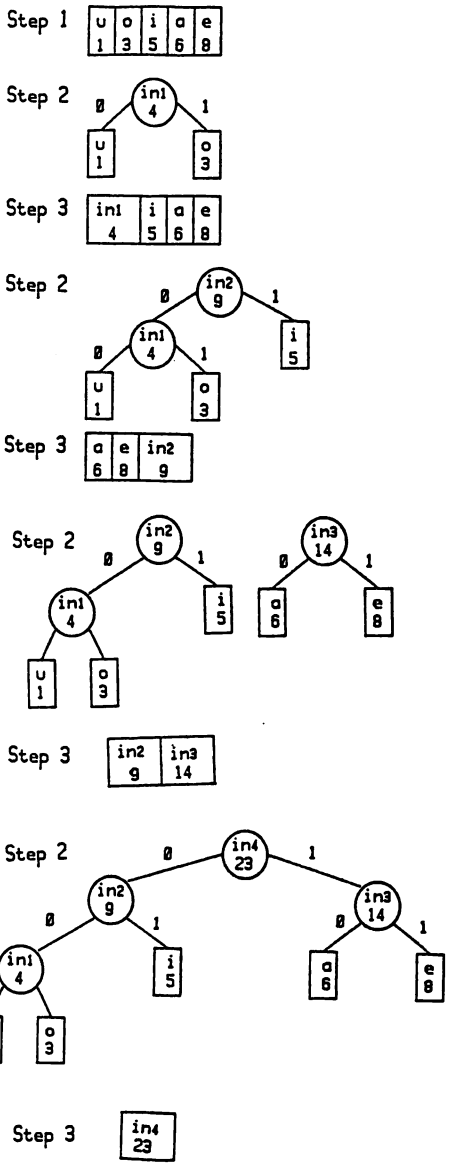

Step 1
u 1
o 3
i 5
a 6
e 8
Step 2
in1 4
0
1
u 1
o 3
Step 3
in1 4
i 5
a 6
e 8
Step 2
in2 9
0
1
in1 4
i 5
0
1
u 1
o 3
Step 3
a 6
e 8
in2 9
Step 2
in2 9
0
1
in1 4
i 5
in3 14
0
1
a 6
e 8
u 1
o 3
Step 3
in2 9
in3 14
Step 2
in4 23
0
1
in2 9
in3 14
0
1
0
1
in1 4
i 5
a 6
e 8
u 1
o 3
Step 3
in4 23

Figure IV.48

- the size of the original (ASCII) text was 23 bytes;
- the size of the encoded text will be
  8*2 + 6*2 + 5*2 + 3*3 + 1*3 = 50 bits = 7 bytes.
- the Huffman Code has thus saved us 16 bytes, or about 70% of the original size.

IV.7.2.2. Building the Tree

In order to build the Huffman Tree from the frequency table, the following steps need to be taken:

- *Step 1*
  all characters in the frequency table are sorted in ascending order into a list. That is, the character with the lowest frequency is at the head of the list;
- *Step 2*
  take the first and second elements of the list, create a new internal node, set its left branch to the first element, its right branch to the second element, and mark it with the sum of the frequencies of the branches;
- *Step 3*
  remove the first and second elements from the list, and insert the newly created internal node in the list in such a way that the resulting list is still sorted in ascending order;
- *Step 4*
  if the number of elements in the list is greater than one, goto Step 2, otherwise we are finished.

We will follow this process in detail with our frequency table.

IV.7.2.3. Encoding and decoding

As soon as we have finished with the construction of the Huffman Tree, the following steps need to be taken:

- determine the Huffman Code for each character (its length and its bit pattern);
- encode the original text;
- save the Huffman Tree in some convenient form as it is needed later to decode our text.

Finally, we present the algorithm to decode our text:

- *Step 1*
  start at the root of the Huffman Tree;
- *Step 2*
  take the next bit from the encoded text and go to the left (right) subtree node if it is a "0" ("1");
- *Step 3*
  if the new node is an internal node, go to Step 2; if it is a leaf node, print the corresponding character and go to Step 1, unless it is an end-of-message character, in which case we quit.

IV.8. Hashing

In many computer science applications implementation of some sort of a dictionary is needed. For instance a Basic interpreter must store keywords and user defined identifiers in a data structure such that it can search efficiently for a name encountered in the Basic program.

One way to implement a dictionary is an array of buckets. In every bucket one word of the dictionary can be stored. To speed up the searching process we will have to keep the buckets ordered (for example in lexicographical order). A disadvantage of this method is that we have to know in advance the maximum number of different words encountered in Basic programs. If the number of words becomes large searching for a specific word will take a relatively large amount of time.

Fortunately we can improve search times by using the important and widely used technique called *hashing*. If we can determine a unique number for every word encountered (the *key*) and if we were to know in advance that the keys are distinct integers from 1 to N we could store word w with key k at position k in an array (the *table*), ready for immediate access.

Clearly the first step in the hashing technique is to calculate the key of a word according to the "hash function". This function transforms a word into an integer key. There is one drawback, however: no hash function transforms every word encountered into a unique key. So what should we do when two distinct words are transformed into the same key value?
Principally there are two methods to deal with this problem:

1) open hashing
2) closed hashing.

IV.9. Open hashing

Our table consists of so called bucket headers pointing to lists of little buckets, each containing one word, see Figure IV. 49

A bucket consists of two elements; the first one is the word hashed to this specific bucket list; the second one is anotherbucket header which points to the rest of the bucket list.

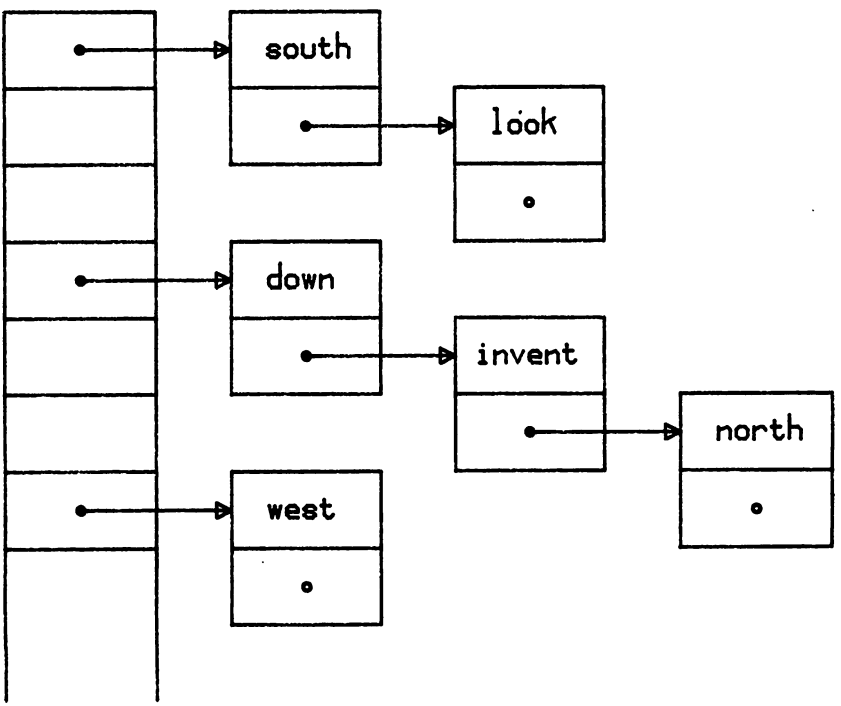


Figure IV.49

If the hash function distributes all words randomly over the table, every bucket list will be relatively short and will on the average have the same length. Thus searching such a list can be performed quickly. Note also that we do not need to know in advance the total number of different words encountered. We simply add another bucket ("dynamically") to one of the

bucket lists. A disadvantage of this method is that we need extra memory to store all the bucket headers.

A good example of a hash function is one that sums all character codes of a word and divides the resulting value by B (the size of the table), taking the remainder, which is an integer value in the range from 0 to B−1. Usually we choose a prime number for the value B.

IV.10. Closed hashing

Closed hashing keeps the words of the dictionary in the table, rather then using bucket headers and lists. This saves memory space. As a consequence, we can put just one word in every table entry. What should we do with two words hashing to one key value? We use a *rehash* strategy. If we try to place word w in bucket table[h(w)] according to hash function h and we find it already contains word w'(different from w), we are confronted with a so called *collision.* With a rehash function r we calculate an alternative position starting from location h(w). We calculate r and try to store w in a bucket until we find an unused one. If there are no empty buckets left, our table is full, and we cannot insert item w.

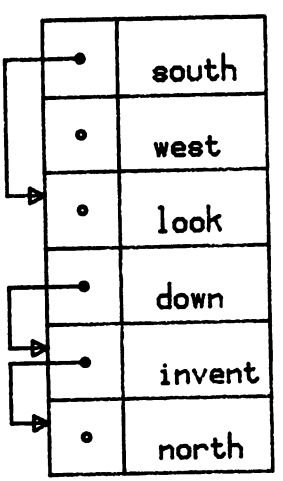
The simplest rehash function, called *linear hashing,* just tries the next bucket, that is

suppose h(w)=x *then* r(x) = (x+1) *mod* B.

Performance measurements have shown that this rehash function drastically increases search times when the table gets nearly full. In linear hashing different words are hashed too close (this phenomenon is called "clustering"). To avoid clustering we need a more "random" rehash function. Two examples are listed in Figure IV.50.

r(x) = B − 2 − (x *mod* (B-2))
r(x) = (x + n^2) *mod* B,
with n increased by 1 every time the rehash fails to find an empty bucket.

Figure IV.50



APPENDIX A

Syntax of the ADL

We will now give the syntax description for the Adventure Description Language in BNF notation.

```
<program> ::=
    <verbdef> <program> | <verbdef>

<verbdef> ::=
    <transitive verb def> | <nontransitive verb def>

<nontransitive verb def> ::=
    <verb> <statements> END

<transitive verb def> ::=
    <verb> <object> <object condition> <statements>
  { NEXT <object> <object condition> <statements> }
    END

<object condition> ::=
    LOCAL | INVENT | SKIP

<statements> ::=
    <statement part> <statement> | <statement part>

<statement part> ::=
    <if statement> | <case statement> | <action
procedure>

<if statement> ::=
    IF <condition> THEN <statements>
 { ELIF <condition> THEN <statements> }
    <else part>
    FI

<else part> ::=
    ELSE <statements> | <empty>
```

```
<condition> ::=
    <simple condition> AND <condition> |
    <simple condition> OR <condition> |
    <simple condition>

<simple condition> ::=
    NOT <condition procedure> | <condition procedure>

<case statement> ::=
    CASE <variable>
  { OF <range> : <statements> }
    <default part>
    ESAC

<default part> ::=
    DEFAULT <statements> | <empty>

<range> ::=
    <integer value> { , <integer value> }

<empty> ::=
```

Examples of the following categories are given:

```
<verb> ::=
    take | look | west ......

<object> ::=
    diamond | axe | money ......

<variable> ::=
    CURLOC | TOILET | DUMMY ......

<integer value> ::=
    23 | 1985 | 6 ......

<action procedure> ::=
    MESSAGE | ADD | SETVAR ......

<condition procedure> ::=
    EQUAL | TESTCL | OBJSTR ......
```

APPENDIX B

The complete red button game listing

B1. The transition table:

```
1  in  0
2  e   3 n 4
3  w   2 s 0 e   8 n 11
4  w   5 n 6 s   2 e 10
5  e   4
6  s   4
7  n   3
8  w   3 n 9 out 3
9  s   8
10 w   4 ALLMOV 0
11 s   3 in 0 ALLMOV 0
12 out 11
```

B2. The dictionary:

```
SUCCESS
LOGIN
LOADED
PAYED
COFFEE
TOILET
CODE1
CODE2
DUM2
[ end of variables
in          1
enter       1
out         2
n           3
north       3
s           4
south       4
```

```
w           5
west        5
e           6
east        6
[ end of move verbs
look        1
help        2
login       3
quit        4
invent      5
run         6
load        7
logout      8
pacman      9
alarm1      20
alarm2      21
alarm3      22
[ end of intransitive verbs
take        1
drop        2
give        3
mount       4
unmount     5
ask         6
call        7
insert      8
drink       9
buy         10
open        11
[ end of transitive verbs
a  receptionist    1   FRIEND    2   1
   money           2   AID       1   2
a  coin            3   AID     255   3
an engineer        4   FRIEND    6   4
a  manager         5   FRIEND    8   5
a  tape            6   AID      12   6
a  magtape         6   AID      12   6
a  floppy          7   AID       9   7
a  door            8   NOCLASS  11   8
   coffee          9   AID     255   9
   code           10   NOCLASS 255 255
[ end of objects
```

B3. The various string sections:

#1 I am not deaf sir !
#2 You do not have any money
#3 You didn't use the last coin I sold you
#4 There is no coffee machine here

#5 Well it does happen sometimes:
The machine didn't eject a cup and you see your coffee pouring down the drain
#6 OK !
#7 Drinking to much coffee is not good for you
#8 You feel much more awake; it certainly will benefit you in your attempt to shut the system
#9 The coffee has a side-effect: you have to go to the toilet...
#10 You didn't go to the toilet and it ain't much use walking around as a COMPUTER SPECIALIST with wet trousers
#11 The only one who drinks coffee apart from you is the manager
#12 No time for coffee boy, I'm busy
#13 The manager grabs your coffee, swallows it and turns into a human being again
#14 From the possible 46 points you have scored:
#15 What's even better you saved the world !! You are really the master adventurer
#16 Your performance was a disaster,
#17 You are near the solution,
#18 fortunately Cruise Missiles don't go off that easily. So give it another try
#19 The manager does not react, he is too sleepy
#20 He responds; the low code part is:
#21 He responds; the high code part is:
#22 There is nobody with authorisation here
#23 You can use the system only at the main terminal
#24 Illegal command
#25 Low code number please:
#26 High code number please:
#27 Welcome to Impossible Big Machine
–chess has been removed from the system due to performance degradation
#28 You had a 0.3 % chance, and you missed
#29 Illegal user code
#30 You can select the following possibilities:

```
  0 : shutdown    1 : BASIC
  2 : FORTRAN     3 : PASCAL
```

Select menu item :
#31 Inserting floppies requires a disk unit, stupid
#32 Wiseguy
#33 Dropping computer supplies is quite harmful, before the medium touches the ground a student catches it and runs away
#34 It is not mounted
#35 MCL-> Not logged in
#36 On a mainframe ? Tut tut
#37 There is no time for processor consuming tasks
#38 You have an higher education, I suppose
#39 There is no medium here to load from
#40 MCL-> Task not yet in system
#41 There is no tape unit here
#42 The tape moves, but a load fails, something else is needed

#43 The floppy clicks, but nothing interesting happens
#44 shutdown is loaded in main memory, it's getting hot in here. It is a one time load only, because the machine eats both the floppy and the magtape...
#45 It is not movable
#46 Can't you read ? THE DOOR IS CLOSED !
#47 You are too sleepy to walk through the computer room
#48 The door is already open
#49 You receive a coin for the coffee machine
#50 It's a relief to get rid of all the coffee
#51 You can buy something here, but you have to attract the attention of the receptionist first
#52 This person knows something you would like to know
#53 You have to login and to run a program that is not yet on the computer..
#54 The door is computer controlled, authorise yourself
#55 No help for this location
#56 Sorry you didn't make it in time
#57 It is still mounted
#58 The receptionist is too busy playing pacman
#59 No one with coins here
#60 Number of minutes left:
#61 You are not authorised by the computer
#62 After you have played pacman for about 5 minutes you stop.
#63 You can't play pacman here
$ end of messages
#0 You are currently holding the following:
#1 You are in front of a building reading 'Hexagon Computer Centre. Some lunatic has pushed the red button and all systems run automatically, the only one to stop this madness seems to be you !! But hurry you have only about 70 turns.....
#2 You are in the hall of the computer centre. To the north and east are corridors
#3 You're in a corridor, you can go in several directions. The sound of computer equipment can be heard
#4 You're in a corridor, you can go in several directions. The sound of computer equipment can be heard
#5 You are in the cantine, it has a coffee machine that requires coins
#6 You are in the maintenance room
#7 You are on the toilet
#8 You are in the room of the system manager
#9 You are in a small supply room
#10 You are in the computer room. There is a floppy unit and a magtape unit here
#11 You are in the computer room. In the corner is a little door. The main terminal is here
#12 You are in a little secret bunker
$ end of location descriptions
#1 A receptionist is playing pacman
#2 There is money here
#3 There is a coin here for the coffee machine

```
#4 An engineer is repairing a magtape unit
#5 The manager is sitting in his chair
#6 There is a red magtape here
#7 There is a dusty floppy disk here
#8 The door is closed
#9 There is hot coffee here
#64 The receptionist is at your service
#65 The manager is hacking away at his terminal like a madman
#66 A magtape is mounted
#67 A floppy is inserted
#68 The door is open
$ end of object descriptions
```

B4. The syntax:

```
s           IF EQUAL TOILET 1 THEN
               ADD TOILET 1
               ADD SCORE 5
               MESSAGE 50
            FI
            SETVAR CURLOC 7
END
in
            IF EQUAL CURLOC 1 THEN
               ALARM alarm3 10
               SETVAR CURLOC 2
               SETRAN CODE1 99
               SETRAN CODE2 99
            ELIF OBJSTR door 8 THEN MESSAGE 46
            ELSE SETVAR CURLOC 12
            FI
END
ALLMOV      IF EQUAL COFFEE 1 THEN
               IF EQUAL CURLOC 10 THEN SETVAR CURLOC 11
                                  ELSE SETVAR CURLOC 10
               FI
            ELSE MESSAGE 47
            FI
END
alarm1
            CHOSTR receptionist   1
END
alarm2
            CASE TOILET
             OF 0 : MESSAGE 9
                    ADD TOILET 1
                    ALARM alarm2 6
             OF 1 : MESSAGE 10
                    REPROC quit CLAY
             OF 2 : SETVAR TOILET 0
```

```
            ESAC
END
alarm3
            IF SMALLER NTURNS 70 THEN
               SETVAR DUMMY 70
               SUB DUMMY ?NTURNS
               MESSAGE 60
               PRINT DUMMY
               ALARM alarm3 10
            ELSE
               MESSAGE 56
               REPROC quit CLAY
            FI
END
quit
               MESSAGE 14
               PRINT SCORE
               IF EQUAL SUCCESS 1 THEN MESSAGE 15
               ELSE IF SMEQUAL SCORE 25 THEN MESSAGE 16
                    ELSE MESSAGE 17
                    FI
                    MESSAGE 18
            FI
            QUIT
END
look
            LISTLOC ?CURLOC
END
invent
            LISTLOC 0
END
help
            CASE CURLOC
             OF  2    : MESSAGE 51
             OF  6, 8 : MESSAGE 52
             OF 11    : MESSAGE 53
             OF 12    : MESSAGE 54
             DEFAULT     MESSAGE 55
                         ADD SCORE 2
            ESAC
            SUB SCORE 2
END
logout
            IF NOT EQUAL CURLOC 11 THEN MESSAGE 23
            ELIF SMALLER LOGIN 4 THEN MESSAGE 35
            ELSE SETVAR LOGIN 0
                 SUB SCORE 10
                 MESSAGE 6
            FI
```

```
END
login
            IF NOT EQUAL CURLOC 11 THEN MESSAGE 23
            ELIF NOT SMALLER LOGIN 4 THEN MESSAGE 24
            ELSE MESSAGE 25
                 READI DUMMY
                 MESSAGE 26
                 READI DUM2
                 IF EQUAL DUMMY ?CODE1 AND
                    EQUAL DUM2 ?CODE2
                 THEN MESSAGE 27
                      SETVAR LOGIN 4
                      ADD SCORE 10
                 ELSE ADD LOGIN 1
                      IF EQUAL LOGIN 3 THEN
                         MESSAGE 28
                         REPROC quit CLAY
                      ELSE MESSAGE 29
                      FI
                 FI
            FI
END
load
            IF NOT EQUAL CURLOC 11 THEN MESSAGE 23
            ELIF SMALLER LOGIN 4 THEN MESSAGE 35
            ELIF NOT OBJSTR tape 66 THEN
                 IF NOT OBJSTR floppy 67 THEN MESSAGE 39
                 ELSE MESSAGE 43
                 FI
                 ELSE IF NOT OBJSTR floppy 67 THEN
                                                MESSAGE 42
                      ELSE SETVAR LOADED 1
                           CHLOC floppy 255
                           MESSAGE 44
                           CHLOC tape 255
                           ADD SCORE 5
                      FI
                 FI
END
run
            IF NOT EQUAL CURLOC 11 THEN MESSAGE 23
            ELIF SMALLER LOGIN 4 THEN MESSAGE 35
            ELSE MESSAGE 30
                 READI DUMMY
                 IF NOT SMALLER DUMMY 4 THEN MESSAGE 32
                 ELIF EQUAL DUMMY 0 THEN
                      IF EQUAL LOADED 1 THEN
                         SETVAR SUCCESS 1
                         REPROC quit CLAY
```

```
                    ELSE MESSAGE 40
                    FI
               ELSE ADD     DUMMY 35
                    MESSAGE ?DUMMY
               FI
          FI
END
pacman
          IF EQUAL CURLOC 11 THEN MESSAGE 62
                                  SUB NTURNS 5
          ELSE MESSAGE 63
          FI
END
ask code  SKIP
            IF PRESENT manager THEN
               IF OBJSTR manager 5 THEN MESSAGE 19
               ELSE SETRAN CODE1 99
                    MESSAGE 20
                    PRINT CODE1
               FI
            ELIF PRESENT engineer THEN
                 SETRAN CODE2 99
                 MESSAGE 21
                 PRINT CODE2
            ELSE MESSAGE 22
            FI
END
give coffee INVENT
          IF NOT PRESENT manager THEN MESSAGE 11
          ELIF OBJSTR manager 65 THEN MESSAGE 12
          ELSE MESSAGE 13
               CHOSTR manager 65
               ADD SCORE 5
               CHLOC coffee 255
          FI
END
call receptionist LOCAL
          IF OBJSTR receptionist 64 THEN MESSAGE 1
          ELSE CHOSTR receptionist 64
               ALARM alarm1 3
          FI
END
buy coin  SKIP
           IF NOT EQUAL CURLOC 2 THEN MESSAGE 59
           ELIF OBJSTR receptionist 1 THEN MESSAGE 58
           ELIF NOT HOLDING money THEN MESSAGE 2
           ELIF NOT OBJLOC coin 255 THEN MESSAGE 3
           ELSE CHLOC coin 0
                ADD PAYED 1
```

```
                    IF EQUAL PAYED 4 THEN CHLOC money 255
                    FI
                    MESSAGE 49
               FI
END
insert coin INVENT
               IF NOT EQUAL CURLOC 5 THEN MESSAGE 4
               ELSE SETRAN DUMMY 99
                    MESSAGE 6
                    IF SMEQUAL DUMMY 70 THEN
                       CHLOC coffee 5
                    ELSE MESSAGE 5
                    FI
                    CHLOC coin 255
               FI
NEXT           floppy INVENT
               IF NOT EQUAL CURLOC 10 THEN MESSAGE 31
               ELSE MESSAGE 6
                    CHOSTR floppy 67
                    CHLOC floppy 10
               FI
END
drink coffee INVENT
               IF EQUAL COFFEE 1 THEN MESSAGE 7
               ELSE CHLOC coffee 255
                    MESSAGE 8
                    ADD SCORE 5
                    ALARM alarm2 6
                    SETVAR COFFEE 1
               FI
END
mount tape INVENT
               IF NOT EQUAL CURLOC 10 THEN MESSAGE 41
               ELSE CHOSTR tape 66
                    CHLOC tape 10
               FI
END
take CLAY      LOCAL
               IF NOT TESTCL CLAY AID THEN MESSAGE 45
               ELIF LASTOBJ tape AND OBJSTR tape 66 THEN
                                                    MESSAGE 57
               ELSE CHLOC CLAY 0
                    IF LASTOBJ floppy THEN CHOSTR floppy 7
                    FI
                    MESSAGE 6
               FI
END
drop CLAY      INVENT
               IF LASTOBJ tape OR LASTOBJ floppy THEN
```

```
            MESSAGE 33
            SETRAN DUMMY 12
            CHLOC CLAY ?DUMMY
        ELSE MESSAGE 6
             CHLOC CLAY ?CURLOC
        FI
END
open door   LOCAL
        IF OBJSTR door 68 THEN MESSAGE 48
        ELIF SMALLER LOGIN 4 THEN MESSAGE 61
        ELSE CHOSTR door 68
             ADD SCORE 5
        FI
END
unmount tape LOCAL
        IF NOT EQUAL CURLOC 10 THEN MESSAGE 23
        ELIF NOT OBJSTR tape 66 THEN MESSAGE 34
        ELSE CHOSTR tape 6
             CHLOC tape 0
        FI
END
```

APPENDIX C

The game of long forgotten Arabia

C.1. The transition table

1 ALLMOV 0			
2 d	3 in	3 ALLMOV 0	
3 out	2 d	4 in	4
4 out	3 u	3	
5 u	6 d	7	
6 d	5		
7 u	5 w	8 s	14 e 31 n 7
8 w	9 in	9 s	12 e 7
9 in	10 e	8 out	8
10 u	0 out	9	
11 ALLMOV 1			
12 n	8 in	0	
13 out	0		
14 n	7 out	0 s	0
15 in	14 s	16	
16 n	0 ALLMOV	17	
17 w	0 ALLMOV	16	
18 e	17 w	19	
19 n	20 e	18 s	21 w 20
20 n	21 e	22 s	23 w 20
21 n	19 e	22 s	21 w 22
22 n	22 e	19 s	20 w 23
23 n	22 e	24 s	23 w 21 in 0
24 n	24 e	24 s	24 w 25
25 e	23 w	26	
26 e	25 in	27	
27 out	0 d	0	
28 d	0 u	27 e	29
29 ALLMOV 29			
30 u	0		
31 e	32 out	32 w	7 in 7

```
32 e        0 in       31 w       31
33 in       0 w        32
34 l       35 r        38 out     33
35 s       34 in       34 n       36 u  41
36 in      37 s        35
37 up       0 out      36 ALLMOV 37
38 ALLMOV   0
39 d        0 up       39 ALLMOV  0
40 u        0
41 d       35
42 u        0
```

C.2. The dictionary

```
  DIGGED
  FILLED
  LIGHTED
  PAYED
  ROPED
  PRICES
  TRYEAST
  DIR
  FLY
  WALK
[ end of variables
  e                1
  east             1
  w                2
  west             2
  n                3
  north            3
  s                4
  south            4
  u                5
  up               5
  d                6
  down             6
  in               7
  enter            7
  exit             8
  out              8
  nw               9
  ne              10
  sw              11
  se              12
  left            13
  l               13
  right           14
  r               14
[ end of move verbs
```

```
  b              1
  back           1
  invent         2
  quit           3
  help           4
  dig            5
  goto           6
  look           7
  swim           8
  sit            9
  alarm1        10
  alarm2        11
[ end of nontransitive verbs
  take           1
  drop           2
  open           3
  unlock         3
  close          4
  lock           4
  kill           5
  ask            6
  push           7
  fill           8
  give           9
  pay            9
  light         10
  steal         11
  eat           12
  drink         13
  brush         14
  polish        14
  examine       15
  tie           16
[ end of transitive verbs
  bread          1 AID        1         1
  Judas          2 FRIEND     2         2
  a knife        3 AID        2         3
  a shovel       4 AID        4         4
  a bottle       5 AID        7         5
  water          6 AID      255         5
  a guard        7 ENEMY     10         7
  money          8 AID       10         8
  a purse        8 AID       10         8
  Shaitan        9 ENEMY     11         9
  a lady        10 FRIEND    13        10
  a woman       10 FRIEND    13        10
  a lamp        11 AID       12        11
  a sentry      14 ENEMY     14        14
  a sign        15 NOCLASS   18        15
```

```
 a rope          16 AID       22        16
 jewels          17 TREASUR   30        17
 a case          17 TREASUR   30        17
 a jewelcase     17 TREASUR   30        17
 a door          18 NOCLASS    1        18
 a door          19 NOCLASS    2        18
 a brick         20 SPECIAL    1        20
 a body          21 NOCLASS  255        21
 a body          22 NOCLASS  255        21
 a crown         23 TREASUR   40        23
 score           24 NOCLASS  255         1
 a towngate      25 SPECIAL   14        24
 a gate          25 SPECIAL   14        24
 a rug           26 AID      255        26
 a carpet        26 AID      255        26
 a helmet        27 TREASUR   41        27
 a sword         28 TREASUR   42        28
 a scimitar      28 TREASUR   42        28
[ end of objects
```

C.3. Strings

#1 I don't see anything like it here
#2 You do not carry it
#3 OK !
#4 It is not movable
#5 You can use words like ''n,s,e,w,nw,ne,sw,se,l,r,u,d''
or ''look,back,quit,take object,drop object''
or ''give object,examine object etc.etc.''
#6
Your adventure has ended. See you next time .....
You have reached a score, which is calculated according to the actions performed, the locations visited and the number of commands given.
Out of the possible score of 156 points your score is:
#7 You cannot give that
#8 I have lost every sense of direction
#9 There is no door here
#10 There is no path in that direction
#11 You don't have the necessary equipment
#12 The door is already closed
#13 As you push against the brick a large part of the wall collapses and a hole is created.
At the other side is another cell with a man in it.
He calls himself Judas and asks you to come over to his cell to help with digging a tunnel out of this prison ....
#14 Judas won't let you come into his cell unless you pay him
#15 Judas invites you to come into his cell to see the tunnel yourself

#16 Boy, you are a crude adventurer ! Well ....
#17 With a soft splash it falls into the pool and vanishes out of sight!
#18 You are insane, eh boy !?
#19 As you come into open air the tunnel collapses behind you. Unfortunately the shovel is buried too
#20 There is no open water around here
#21 Your bottle is already full of water
#22 The guard angrily grabs you and transports you to a safe place ...
#23 That would be an easy way to get rid of enemies !
#24 Secretly and succesfully you steal it from the guard
#25 What! Are you a thief ?
#26 A woman speaks to you the following words:
"Although you have changed a lot I recognize you as I was your old nurse. You are the only heir of the late khalif of Uzurkistan. I will help you to regain the throne of the khalifate by giving you information: you must find the family treasures that where lost in the fightings, which broke out after your father died.

The evil wizard Shaitan has seized power and reigns in the palace until someone claims the throne, with the treasures as proof of his or her rightfull heirage of the throne. Go to the hills and bring back the jewelcase of Hassan the Great."
#27 The woman speaks to you the following words:
"You have fulfilled this very difficult task splendidly. Now you will have to trace the famous Royal Crown with the 3 emeralds. But beware! The wicked Shaitan has spotted you in his cristal ball. He will not rest until he has locked you up safely, or ... still better has found the Crown for himself, because in that way no one can ever prove to be the rightful heir of the throne!!!"
#28 Shaitan growls at you:
"How dare you to disturb me in my palace, you son of a dog !!! Guards, throw him in the prison for the rest of his life !!!"
#29 Shaitan speaks to you:
"Oh, what a poor man I am ! This is the rightful heir of the throne before me ! Forgive oh master, thy unworthy slave for all his sins !!"
#30 You dared to enter the cave of the great cavebear. The bear is awakened by your approach and furiously chops your head off !!
#31 You didn't have any water with you, so ... overcome by thirst you finally collapse and die in the glare of the sun.
#32 As you jump into the water a slimy creature with sixteen tentacles grabs your legs and arms and pulls you under the dirty water. That was your last heroic adventure in this game !
#33 You cannot swim here.
#34 You are a fool, to jump into a pit ! You broke all your bones !!!
#35 As you stumble in the dark you fall into a pit ... and break all your bones !!!
#36 The woman speaks to you the following words:
"You have gathered all treasures, go to the palace and claim the throne !"

#37 I hear hoofclatter somewhere behind you, I think you are being followed !?
#38 Your accumulated score is at the moment:
#39 There is nobody around who knows your score
#40 During your stroll in the desert you drank up the water in the flask.
#41 The lamp is already lighted
#42 Unfortunately you have used up the power of the lamp
#43 As you exit your lamp goes dark
#44 The spirit of the lamp is too tired to come and help you, sorry !
#45 The woman speaks to you the following words:
"Why don't you have the jewelcase with you ?
You really need it if you want to claim the throne !"
#46 What is this, a joke ?? You must have done everything wrong !?!
(To get the next higher rank you must score 10 points)
#47 You are clearly a novice in this game !
(To get the next higher rank you must score 35 points)
#48 You are learning, keep going !
(To get the next higher rank you must score 70 points)
#49 Well, that wasn't too bad, now was it ?!
(To get the next higher rank you must score 100 points)
#50 Jolly good work, but still not perfect !
(To get the next higher rank you must score 145 points)
#51 You are one of the best Adventure Game players I have ever seen !!!
But, ------
to be Master of the Adventurers you must score the maximum.
#52 This is a friend, and may be helpful in some way
#53 This is a not so helpful person, in fact may be hostile to you
#54 This is an aid, if you know how to use it
#55 This is a treasure worth a lot
#56 This is a special object in this game
#57 This is no special object in any way
#58 "In my cristal ball I can see that Shaitan knows of your escape and has send a hunting party after you, so hurry !!!"
#59 You are about to enter the harem of this house, when a eunuch appears: he cuts your head off.
#60 You flew some time and landed afterwards. You stepped off the flying carpet.
#61 You cannot fly like an eagle ... so you crash down and break your neck !
#62 Never trust a djinn. A lion growls and storms out of the bushes launching a fierce attack on you.
#63 Badly injured you can barely escape the monster.
#64 A djinn appears suddenly before you ! He is telling a story about crowns and diamonds with glittering eyes. He suggests to you to enter the wood and follow the main path. Afterwards the djinn disappears in a cloud of yellow mist.

#65 As nothing interesting happens you stand up again.
#66 You cannot sit on the carpet and carry it at the same time !

#67 Why would you sit on the carpet and loose time by doing nothing ??
#68 **********
You are the Master of Adventure Games !!!
****
#69 The cruel warriors of Shaitan on their black horses have finally located your whereabouts and kill you on the spot !!!
#70 You tied the rope around some rocky stones. $ end of messages
#0 You are currently holding the following:
#1 You are in a prison cell built out of rough cobblestones and cement.
#2 You are in a cell. It seems that someone has started digging a tunnel.
#3 This is the entrance of the tunnel running further and further. The construction looks rather dangerous!!
#4 This is the end of the tunnel. You are standing before a wall of sand and debris.
#5 You stand on a sloping hillside near a heap of sand and rock.
#6 You are on top of the hill. A splendid view of the surrounding terrain is offered to you! Around the hill lies a city. To the west a large palace-like building can be seen. To the south a long and dusty road runs to the horizon.
#7 You are at the foot of a medium sized mountain hill. Roads from east to west and north to south are crossing here.
#8 You are on a big marketplace before the palace of the khalif.
#9 This is a large entrance hall. In the middle is a nicely carved fountain. Through a large door you can go further into the palace.
#10 You are in a marble hall with a large staircase.
#11 You are standing in the middle of the large throne-hall. The walls are of priceless gold.
#12 You are in a dead alley. A small house has its door invitingly open.
#13 Inside the little house it is shadowy dark.
#14 You are at the town gate.
#15 You are on a long road near the town gate.
In the far distance you can see sand dunes: the desert.
#16 You are in a desert. In the distance to the north a town can be seen.
#17 You are in a desert. In the distance to the west you can see hills and rocks.
#18 This is a curving slope with small rocks lying around. In the distance sand dunes can be seen: the desert.
#19 You are in the Hills of No Return
#20 You are in the Hills of No Return
#21 You are in the Hills of No Return
#22 You are in the Hills of No Return
#23 You are in the Hills of No Return
There is a cave in the hillside before you. it is quite smelly here!
#24 You are in the Hills of No Return
#25 You are walking east-west along a narrow winding path. To your left and right you can see a dark marshy pool. It is gobbling slightly in a weird way.
#26 You are before the entrance of a cave in the hillside.
#27 The entrance of the cave is illuminated by the light from the outside.

Further on it is rapidly darkening as the cave slopes down. If I were you I would make some light.
#28 You have stopped just in time: there is a pit right before your feet. There seems to be a small crack in the rock to the east.
#29 You are lost in a maze of caverns and tunnels !!!
#30 You are at the bottom of the pit.
#31 You are at the east town gate, which is open.
#32 This is a road near the town gate running to a large wood in the east.
#33 You are in front of a large forest, a small path enters it here.
#34 You are walking along a path in a dark and large forest. There is a forking path here, going left and right just ahead.
#35 This is the edge of the wood. A pathway goes north. In the distance you can see a small spiral standing out against the sky.
#36 You are about to enter a little white palace built entirely out of marble.
#37 It is pitch dark here.
#38 You are following various forest paths.
#39 Here the road ends abruptly.
A steep hillside at your left is clearly unclimbable. Before you is a frightening deep abyss. This is a dead end!
#40 You are in a ravine with towering walls all around you.
#41 You are on top of the hill. A splendid view takes your breath away. There is a dangerous looking path going down on the other side of the hill.
#42 You are in a valley with high cliffwalls surrounding you. The only way to get out seems to be straight up into the air!
#101 You are in a prison cell built out of rough cobblestones and cement. There is an opening in one wall: it is large enough to go through.
#137 You are standing in a large hall with rich draperies attached to the walls.
A small winding staircase goes up. A sign near you reads:
>> No trespassing for others than the owner of the house!
<< $ end of location strings
#1 A piece of bread is lying in the dust on the floor.
#2 Judas is standing in the corner of his cell.
#3 There is a knife lying on the ground.
#4 There is a shovel before your feet.
#5 An empty bottle lies somewhere to the right of you.
#6 There is a bottle filled with water.
#7 A royal guard guards the staircase (and his money of course!).
#8 Here lies a purse with money.
#9 The great magister Shaitan is sitting on a throne of carved white ivory.
#10 An old woman with shabby clothes sits behind a table.
#11 A brass lamp is lying here.
#12 The town gate is closed.
#13 The town gate is open.
#14 A sentry is standing guard before the gate.
#15 A sign is standing here reading:

"Do not proceed !!
You are about to enter the Hills of No Return.
Never a human entering this place has returned alive!"
#16 A long piece of rope is lying here.
#17 A fine jewelcase with golden strips is lying on the ground.
#18 The cell door is to your left.
#19 A royal guard guards the staircase, he looks angry because somebody stole his money.
#20 There is a loose brick in the wall.
#21 A dead body is sprawled on the ground.
#23 The famous Royal Crown with the three emeralds is lying at your feet.
#24 Sadly enough the town gate is closed.
#25 You're lucky, the town gate is open.
#26 A Persian rug in various warm colors is lying on the ground before you.
#27 A golden crownlike helmet with various precious stones set into its brim is here.
#28 There is an ancient Turkish scimitar with a silver hilt.
#29 The rope is hanging down into the pit, where it just touches the ground.
$ end of object descriptions

C.4. Syntax of commands

```
alarm1      MESSAGE 37
            SETRAN DUMMY 5
            ALARM alarm2 ?DUMMY
END
alarm2      MESSAGE 69
            REPROC quit CLAY
END
ALLMOV      IF   EQUAL CURLOC 1 THEN
                 IF   TESTCL Judas FRIEND THEN
                      IF PRESENT brick THEN MESSAGE 10
                      ELSE MESSAGE 14
                      FI
                 ELSE SETVAR CURLOC 2
                      SETVAR OLDLOC 1
                      IF EQUAL CURLOC 2 THEN SETVAR
                                                CUROBJ 1
                      FI
                 FI
            ELIF EQUAL CURLOC 2 THEN
                 SETVAR CUROBJ 0
                 SETVAR CURLOC 1
                 SETVAR OLDLOC 2
            ELIF EQUAL CURLOC 38 THEN
                 IF   EQUAL WALK 2 THEN
                      SETVAR WALK 0
```

```
                    IF EQUAL DIR 0 THEN SETVAR CURLOC
                                                    39
                    ELSE SETVAR CURLOC 34 SETVAR DIR 0
                    FI
               ELSE ADD WALK 1
                    SETVAR OLDLOC 38
               FI
          ELIF EQUAL CURLOC 39 THEN
               SETVAR DIR 1
               SETVAR OLDLOC 39
               SETVAR CURLOC 38
          FI
END
up        IF   EQUAL CURLOC 10 THEN
               SETVAR CURLOC 11
               SETVAR OLDLOC 11
               IF   HOLDING jewels AND HOLDING crown
                                                    THEN
                    ADD SCORE 10
                    MESSAGE 29
                    REPROC quit CLAY
               ELSE MESSAGE 28
               FI
          ELIF EQUAL CURLOC 30 THEN
               CHLOC rope 28
               CHCLASS rope AID
               CHOSTR rope 16
               SETVAR CURLOC 28
               SETVAR OLDLOC 28
               SETVAR ROPED 0
               ADD SCORE 5
          ELIF EQUAL CURLOC 37 THEN
               MESSAGE 59
               REPROC quit CLAY
          ELIF EQUAL CURLOC 40 OR EQUAL CURLOC 42 THEN
               IF   EQUAL FLY 1 THEN
                    SETVAR FLY 0
                    SETVAR CURLOC 39
                    SETVAR OLDLOC 39
                    MESSAGE 60
                    CHLOC carpet 39
               ELSE MESSAGE 10
               FI
          ELSE MESSAGE 10
          FI
END
down      IF   EQUAL CURLOC 27 THEN
               IF   EQUAL LIGHTED 0 THEN
                    MESSAGE 35
```

```
                REPROC quit CLAY
           ELSE SETVAR OLDLOC 27
                SETVAR CURLOC 28
                ADD    SCORE 5
           FI
      ELIF EQUAL CURLOC 28 THEN
           IF   EQUAL ROPED 0 THEN
                MESSAGE 34
                REPROC quit CLAY
           ELSE
                CHLOC rope 30
                SETVAR CURLOC 30
                SETVAR OLDLOC 30
                ADD    SCORE 5
           FI
      ELIF EQUAL CURLOC 39 THEN
           IF   EQUAL FLY 1 THEN
                SETRAN DUMMY 3
                SETVAR FLY 0
                ADD    CURLOC ?DUMMY
                SETVAR OLDLOC ?CURLOC
                MESSAGE 60
                CHLOC carpet ?CURLOC
           ELSE MESSAGE 61
                REPROC quit CLAY
           FI
      ELSE MESSAGE 10
      FI
END
in    IF EQUAL CURLOC 12 THEN
           SETVAR CURLOC 13
           SETVAR OLDLOC 12
           IF PRESENT lady THEN
              IF   EQUAL PRICES 0 THEN
                   IF   HOLDING jewels OR PRESENT
                                              jewels
                   THEN SETVAR PRICES 1
                        ADD SCORE 45
                        MESSAGE 27
                   ELSE MESSAGE 26
                        IF NOT TESTCL Judas
                                              SPECIAL
                        THEN MESSAGE 58
                        FI
                   FI
              ELIF EQUAL PRICES 1 THEN
                   IF HOLDING crown OR PRESENT
                                                crown
                   THEN IF   HOLDING jewels
```

```
                                    OR PRESENT jewels
                                  THEN SETVAR PRICES 2
                                       ADD SCORE 45
                                       MESSAGE 36
                                  ELSE MESSAGE 45
                                  FI
                             ELSE MESSAGE 27
                             FI
                        FI
                   FI
ELIF EQUAL CURLOC 23 THEN
     MESSAGE 30
     REPROC quit CLAY
ELIF EQUAL CURLOC 33 THEN
     IF   EQUAL PRICES 1 THEN
          SETVAR CURLOC 34
          SETVAR OLDLOC 33
     ELSE MESSAGE 62
          IF   HOLDING knife THEN
               MESSAGE 63
               SUB SCORE 3
               SETVAR CURLOC 33
          ELSE REPROC quit CLAY
          FI
     FI
ELSE MESSAGE 10
FI
END
out        IF   EQUAL CURLOC 13 THEN
               SETVAR CURLOC 12
               SETVAR OLDLOC 13
               IF   EQUAL LIGHTED 1 THEN
                    SETVAR LIGHTED 2
                    MESSAGE 43
               FI
           ELIF EQUAL CURLOC 14 AND EQUAL PAYED 1 THEN
               IF   NOT TESTCL Judas SPECIAL THEN
                    SETRAN DUMMY 10
                    ALARM alarm1 ?DUMMY
               FI
               SETVAR CURLOC 15.
               SETVAR OLDLOC 14
               ADD SCORE 5
           ELIF EQUAL CURLOC 27 THEN
               SETVAR CURLOC 26
               SETVAR OLDLOC 27
               IF   EQUAL LIGHTED 1 THEN
                    SETVAR LIGHTED 2
                    MESSAGE 43
```

```
            FI
        ELSE MESSAGE 10
        FI
END
s       REPROC out CLAY
END
n       IF EQUAL CURLOC 16 THEN
          IF EQUAL FILLED 1 AND HOLDING bottle THEN
            MESSAGE 40
            SETVAR FILLED 0
            SETVAR CURLOC 15
            SETVAR OLDLOC 15
            ADD    SCORE 5
          ELSE MESSAGE 31
               REPROC quit CLAY
          FI
        FI
END
w       IF EQUAL CURLOC 17 THEN
          IF EQUAL FILLED 1 AND HOLDING bottle THEN
            MESSAGE 40
            SETVAR FILLED 0
            SETVAR CURLOC 18
            SETVAR OLDLOC 18
            ADD    SCORE 5
          ELSE MESSAGE 31
               REPROC quit CLAY
          FI
        FI
END
e       IF     EQUAL TRYEAST 0 THEN
               SETVAR TRYEAST 1
               MESSAGE 64
        FI
        SETVAR OLDLOC 32
        SETVAR CURLOC 33
END
quit    MESSAGE 6
        SETVAR DUMMY ?SCORE
        SUB DUMMY ?NTURNS
        ADD DUMMY 42
        IF SMALLER SCORE ?DUMMY
        THEN SETVAR DUMMY ?SCORE
        FI
        PRINT DUMMY
        IF   SMALLER DUMMY 10 THEN MESSAGE 46
        ELIF SMALLER DUMMY 35 THEN MESSAGE 47
        ELIF SMALLER DUMMY 70 THEN MESSAGE 48
        ELIF SMALLER DUMMY 100 THEN MESSAGE 49
```

```
            ELIF SMALLER DUMMY 140 THEN MESSAGE 50
            ELIF SMALLER DUMMY 156 THEN MESSAGE 51
            ELSE                        MESSAGE 68
            FI
            QUIT
END
goto        READI DUMMY
            IF     NOT EQUAL DUMMY 0 THEN
                   SETVAR OLDLOC ?CURLOC
                   SETVAR CURLOC ?DUMMY
                   SUB     SCORE 1
                   LISTLOC ?CURLOC
                   ELSE MESSAGE 8
            FI
END
look        LISTLOC ?CURLOC
END
invent      LISTLOC 0
END
back        IF   EQUAL   CURLOC ?OLDLOC
            THEN MESSAGE 8
            ELSE SETVAR  CURLOC ?OLDLOC
                 LISTLOC ?CURLOC
            FI
END
help        MESSAGE 5
            SUB SCORE 5
END
dig         IF HOLDING shovel THEN
               IF EQUAL CURLOC 4 THEN
                  IF     NOT SMEQUAL DIGGED 1 THEN
                         SETVAR CURLOC 5
                         SETVAR OLDLOC 5
                         ADD     SCORE 5
                         CHLOC shovel 255
                         MESSAGE 19
                         LISTLOC 5
                   ELSE MESSAGE 3
                         ADD     DIGGED 1
                   FI
               ELSE MESSAGE 3
               FI
            ELSE MESSAGE 11
            FI
END
swim        IF EQUAL CURLOC 25 THEN
                 MESSAGE 32
                 REPROC quit CLAY
            ELSE MESSAGE 33
```

```
            FI
END
sit         IF   PRESENT carpet THEN
                 IF   EQUAL CURLOC 39
                   OR EQUAL CURLOC 40
                   OR EQUAL CURLOC 42
                 THEN SETVAR FLY 1
                      MESSAGE 3
                 ELSE MESSAGE 65
                 FI
            ELIF HOLDING carpet THEN MESSAGE 66
            ELSE MESSAGE 67
            FI
END
open door   SKIP
            IF   PRESENT door THEN MESSAGE 11
            ELSE MESSAGE 1
            FI
END
close door SKIP
            IF   PRESENT door THEN MESSAGE 12
            ELSE MESSAGE 1
            FI
END
take        CLAY LOCAL
               IF  NOT TESTCL CLAY AID
                   AND NOT TESTCL CLAY TREASURE THEN
                   MESSAGE 4
              ELIF EQUAL CURLOC 10 THEN
                   MESSAGE 22
                   SETVAR CURLOC 1
                   SETVAR OLDLOC 1
              ELSE CHLOC CLAY 0
                   IF   TESTCL CLAY TREASURE THEN
                        ADD SCORE 5
                   FI
                   MESSAGE 3
              FI
END
drop        CLAY INVENT
               IF   EQUAL CURLOC 25 THEN
                    MESSAGE 17
                    CHLOC CLAY 255
                    LISTLOC 25
               ELSE CHLOC CLAY ?CURLOC
                    IF TESTCL CLAY TREASURE THEN
                       SUB SCORE 5
                    FI
                    MESSAGE 3
```

```
            FI
END
kill        CLAY LOCAL
            IF   HOLDING knife THEN
                 IF PRESENT Judas OR PRESENT lady THEN
                      MESSAGE 16
                      CHLOC CLAY 255
                      CHCLASS CLAY SPECIAL
                      IF LASTOBJ lady THEN SETVAR
            CUROBJ 1
                      ELSE ADD SCORE 10
                      FI
                      CHLOC body ?CURLOC
                 ELIF TESTCL CLAY ENEMY THEN MESSAGE
            23
                 ELSE MESSAGE 5
                 FI
            ELSE MESSAGE 11
            FI
END
ask score   SKIP
            IF   PRESENT lady THEN
                 MESSAGE 38
                 SETVAR DUMMY ?SCORE
                 SUB    DUMMY ?NTURNS
                 ADD    DUMMY 32
                 PRINT  DUMMY
            ELSE MESSAGE 39
            FI
END
push brick LOCAL
            MESSAGE 13
            CHLOC brick 255
            CHLSTR ?CURLOC 101
            LISTLOC ?CURLOC
            ADD      SCORE 5
END
fill bottle INVENT
            IF   EQUAL CURLOC 9
            OR   EQUAL CURLOC 25 THEN
                 IF EQUAL FILLED 1 THEN MESSAGE 21
                 ELSE SETVAR FILLED 1
                      MESSAGE 3
                 FI
            ELSE MESSAGE 20
            FI
END
drink       CLAY SKIP
            IF   LASTOBJ water OR LASTOBJ bottle THEN
```

```
                IF HOLDING bottle AND EQUAL FILLED 1
                                                    THEN
                   MESSAGE 3
                   SETVAR FILLED 0
              ELIF EQUAL CURLOC 8 OR EQUAL CURLOC 25
              THEN MESSAGE 3
              ELSE MESSAGE 20
              FI
           ELSE MESSAGE 18
           FI
END
give bread INVENT
CHCLASS Judas ENEMY
CHLOC bread 255
MESSAGE 15
NEXT
           CLAY INVENT
             IF EQUAL CURLOC 14 AND LASTOBJ money THEN
                MESSAGE 3
                CHOSTR towngate 25
                SETVAR PAYED 1
                CHLOC  CLAY 255
           ELIF EQUAL CURLOC 13 THEN
                MESSAGE 3
                CHLOC CLAY 13
                IF   TESTCL CLAY TREASUR THEN ADD SCORE
           10
                FI
           ELSE MESSAGE 7
           FI
END
eat bread INVENT
           MESSAGE 3
           CHLOC bread 255
END
light lamp INVENT
           IF   EQUAL LIGHTED 0 THEN
                ADD LIGHTED 1
                MESSAGE 3
           ELIF EQUAL LIGHTED 1 THEN
                MESSAGE 41
           ELIF EQUAL LIGHTED 2 AND EQUAL CURLOC 37
           THEN
                CHLSTR 37 137
                LISTLOC 37
                CHLOC carpet 37
           ELSE MESSAGE 42
           FI
END
```

```
polish lamp INVENT
          MESSAGE 44
END
steal purse SKIP
          IF   PRESENT guard THEN
               MESSAGE 24
               CHOSTR guard 19
               CHLOC purse 0
               ADD   SCORE 5
          ELSE MESSAGE 1
          FI
NEXT
          CLAY SKIP
            MESSAGE 25
END
examine CLAY LOCAL
          IF   TESTCL CLAY FRIEND  THEN MESSAGE 52
          ELIF TESTCL CLAY ENEMY   THEN MESSAGE 53
          ELIF TESTCL CLAY AID     THEN MESSAGE 54
          ELIF TESTCL CLAY TREASUR THEN MESSAGE 55
          ELIF TESTCL CLAY SPECIAL THEN MESSAGE 56
          ELSE                          MESSAGE 57
          FI
END
tie       rope INVENT
          IF   EQUAL CURLOC 28 THEN
               MESSAGE 70
               SETVAR ROPED 1
               CHCLASS rope SPECIAL
               CHOSTR rope 29
               CHLOC rope 28
          ELSE MESSAGE 3
               CHLOC rope ?CURLOC
          FI
END
```

APPENDIX D

Scanner error messages

Apart from the messages listed below, the scanner will, if possible, tell you what it it expected at the point where the error occurred.

Game too large!
the game will not fit into the available memory

Dict?
the dictionary file cannot be found

Only n sections in dict
there must be five sections in the dictionary, separated by a dollar sign on the first position of a separate line

Hashtab?
the user hashtable is full

xxx already exists
you have used an identifier for the second time

Trans?
the transition table file cannot be found

Too many transitions
the maximum number of transitions per location is five

No transitions
the minimum number of transitions per location is one

xxx is isolated
(warning only) there is no transition to this location. Of course the transition may be conditional, and therefore specified in the syntax file

xxx has no further transitions
(warning only) there is a transition to this location, but no transitions out of it have been defined. Again it may be a conditional transition

Syntax?
the syntax file cannot be found

Missing item
you did not give enough parameters to a predefined function

Deref?
you can use the dereference operator only with a variable, not with a number (unlike BASIC)

Digit?
a non-digit was found inside a number

Value too big
the maximum value of an integer is 255

Unexpected item
you've given too many parameters to a predefined function, or made some other error

Unknown item
probably indicates a typing error. Note that the scanner distinguishes between lowercase and uppercase characters

Type?
an identifier with the wrong type was found (for example a number instead of a variable)

Eof?
unexpected end-of-file

Precondition expected
The first line of a verb definition must contain one of the keywords SKIP, INVENT or LOCAL

Statement?
an improper statement was found

xxx has no syntax
(warning only) no syntax desciption for xxx was found in the syntax file

Interpreter error messages

What or who must I xxxx
you get this message if you type a transitive verb without an object to go with it

I don't know the word xxxx
a typing error, or perhaps you forgot to press the 'caps lock' key

What must I do with xxxx
you have typed the name of an object, without a verb to go with it

There is no way xxxx
no transition in the indicated direction can be found, either conditional or unconditional

Impossible
you are trying to do something obviously impossible

You are not carrying xxxx
you want to do something with an object you don't have in your possession

I don't see xxxx (here)
you want to do something with an object that is not present at the current location

A value between 0 and 255 please
the number you just typed was out of range

APPENDIX E

The Screen Editor

The adventure system described in this book requires you to build and modify text files. The BBC Micro has no standard facilities for this, and if you don't already have a word processor ROM, you will need some sort of text editor.

This chapter shows a listing of a simple, but powerful screen editor. A screen editor is a program that displays (part of) a text file on the screen of your television or monitor, and then enables you to change the contents of this text file by moving the cursor over the screen and modifying the displayed text where required, by simply typing a new character or by typing a special editor command.

Typical editor commands include:
- inserting characters or lines
- deleting characters or lines
- moving through the file
- saving the file to disc
- loading a file from disc.

This particular editor takes commands in the form of 'control strokes', i.e. by pressing down the 'ctrl' key, holding it down, and then pressing another key (always a letter).

The possible control strokes are shown in table E1.

The editor works in two different modes, 'normal mode' and 'insert mode'. In normal mode, you can move through the file, move the cursor around on the screen, and change or delete characters.
To insert characters between other ones already there, you have to change to insert mode. In insert mode, the only characters permitted are the 'visible' characters, plus the 'return' and the 'del'. All other ones are ignored.

We shall now look at the control keystrokes in some detail.

CTRL+a enters insert mode. The screen is cleared from the cursor position onward; in the Beeb's memory the rest of the file is swapped out to the end of free memory, making room for new text. Pressing the 'del'

key will remove the last character typed, on the current line. Pressing 'return' moves the cursor to the next line. If this would make the cursor leave the bottom line, the screen is cleared and the previous 10 lines are displayed. If the maximum line length would be exceeded when typing a word, the whole word is moved to the start of the next line. Pressing CTRL+a in insert mode returns you to normal mode. This will fail, however, when the length of the current line, after leaving insert mode, would exceed 40.

CTRL+c exits the editor; page is reset to &1900.

CTRL+f displays the first page of the file.

CTRL+g loads a file from disc. Please read the section on CTRL+z below first before using this command.

CTRL+h cursor home. The cursor is positioned at the character in the upper left hand corner of the screen.

CTRL+j joins the current and the next lines. This command can only be executed when the cursor is positioned immediately after the last character on the current line, i.e. on the (invisible) return character. This character will be deleted, and the two lines will thus be joined. The command fails if the current line length after execution of the command would exceed 40.

CTRL+k kill rest of line. The rest of the current line is deleted, from the cursor position onward.

CTRL+l last page. The last page of the file is displayed.

CTRL+n next page. The next page of the file is displayed.

CTRL+o next half page. The screen 'window' is moved forward by half a page, and the page that starts at the new top line is displayed.

CTRL+p previous page. The previous page in the file is displayed.

CTRL+q previous half page. The screen window is moved half a page backward, and the page that starts at the new top line is displayed.

CTRL+s save file. The editor will ask you for a file name, and the current contents of memory will the be saved to that file. It is up to you to make sure there is enough room for it on your disc.

CTRL+u deletes the current line.

CTRL+z first time load of file. When you load a file for the first time, some of the lines in it may be longer than 40 characters (the maximum line

length in mode 7). These lines will need to be broken up into chunks of 40 characters, otherwise the editor will get confused. It may take quite a while to load a file with this command; if you are sure all the lines in it do not exceed 40 characters, or if the file was saved from a previous edit session, you should use CTRL+g instead.

del delete. Pressing this key will delete the character at the cursor (normal mode) or delete the last character typed (insert mode).

return returns the cursor to the left margin (normal mode) or moves the cursor to the start of the next line (insert mode).

The arrow keys are used to move the cursor on the screen.

Notes on the program

The editor consists of two programs: "EDIT" and "EDIT2". EDIT initialises the resident integer variables used by the editor, assembles some machine code routines from &C00 onwards, and then chains the main program, EDIT2.
It is slightly faster to save the assembled machine code on a file and then *load it. This is done by the following series of commands:

The original EDIT is now called SEDIT, and EDIT loads the machine code directly into page &C.

In order to keep the amount of memory needed by the main program to a minimum, maximum use is made of the resident integer variables. They are:

S% start of text file
E% end of text file
L% current line number on screen
C% current column number on screen
U% address in memory of character in upper left hand corner of screen
Z% amount of free memory left. If memory is about to run out, a high pitched sound is generated whenever you press return.

The array X% is used to store the length of each line on the screen (including the 'return' at the end) and the array Y% stores the cumulative line lengths (this is done to be able to skip to the next (half) page quickly).

Because this book is not concerned with the principles of text editing, we will not give a detailed explanation of how the program works; you may want to find out for yourself. The program has lots of spaces in it to improve readability; if you decide to type it in, you should leave them out wherever possible.

Possible problems

The main program is loaded into memory from &6800 onwards, and uses memory from &1301 to &67FE to store the text being edited (giving a maximum file length of about 21 K).
At &1300 and at E%+1 the character &FF is stored to delimit the text (this is used by various machine code routines to stop when the start or the end of the file has been reached).
If the program crashes with a 'no room' error this means that PAGE has been set to a too high a value. In this case the following steps should be taken:

- type RUN. this will restart the editor.
- now type CTRL+s and save whatever you had before the crash (this may not be possible if you were in insert mode when the crash occurred).
- change line 2170 of "EDIT". First set PAGE to &6700, RUN the editor, and if it crashes again try &6600 and so on, until you find a value that does not crash the editor.

Ctrl	*Action*
a	Toggle normal mode - insert mode
c	Exit editor
f	First page
g	Get file from disc
h	Cursor home
j	Join lines
k	Delete rest of line
l	Last page
n	Next page
o	Next half page
p	Previous page
q	Previous half page
s	Save text to disc file
u	Delete current line
z	First time load of file
<del>	Delete char at cursor (n.m.); delete last char typed(i.m.)
<return>	Cursor to left margin (n.m.); cursor to next line (i.m.)
<arrows>	move cursor on screen

Table E1–editor commands

```
  1 REM! editor initialisation program
  2 REM! ----------------------------
  3 REM! A.F. de Geus, J.H. Jongejan
  4 REM! and A.M. Koelmans 1984
  5 REM! ----------------------------
  6 REM! Program subject to copyright
  7 REM! To run, type "chain edit"
  8 REM! ----------------------------
 10  *TV 255
 20  MODE7
 30  S%=&1301
 40  E%=S%
 50  L%=0
 60  C%=0
 70  U%=S%
 80  ?(S%-1)=&FF: ?S%=&FF: S%?1=13
 90  OSWRCH=&FFEE
100  OSASCI=&FFE3
110  SOURCE=&70
120  DEST=&72
130  NUMBER=&74
140  START=&70
150  FOR I=0 TO 2 STEP 2
160  P%=&C00
170 [OPT I
180
190 \ ---- procedure mc1 ----
```

```
200 \ swap out memory from 'source'
210 \ to 'dest'. Highest address is
220 \ moved first. Starts insert mode.
230
240       LDA NUMBER+1     \ first move
250       CLC              \ to last
260       ADC SOURCE+1     \ partial page
270       STA SOURCE+1
280       LDA NUMBER+1
290       CLC
300       ADC DEST+1       \ do same for dest
310       STA DEST+1
320       LDY NUMBER       \ number of bytes
330       BEQ resth        \ to move
340 .looph1
350       DEY
360       LDA (SOURCE),Y   \ copy last partial
370       STA (DEST),Y     \ page
380       CPY #0
390       BNE looph1
400 .resth
410       LDX NUMBER+1     \ number of pages
420       BEQ finishh      \ to move
430 .looph21
440       DEC SOURCE+1     \ move to next page
450       DEC DEST+1
460 .looph22
```

```
470       DEY             \ next char
480       LDA (SOURCE),Y
490       STA (DEST),Y
500       CPY #0          \ page done?
510       BNE looph22
520       DEX             \ any pages left?
530       BNE looph21
540 .finishh
550       RTS
560
570 \ ---- procedure mc2 ----
580 \ swap in memory from 'source' to
590 \ 'dest'. Lowest address is moved first.
600 \ Ends insert mode; also used to delete
610 \ characters or lines.
620
630       LDY #0
640       LDX NUMBER+1      \ number of pages
650       BEQ rest1         \ to move
660 .loop11
670       LDA (SOURCE),Y
680       STA(DEST),Y
690       INY               \ next char
700       BNE loop11        \ page done?
710       INC SOURCE+1      \ move to next page
720       INC DEST+1
730       DEX               \ any pages left?
```

```
 740      BNE loop11
 750 .rest1
 760      LDX NUMBER          \ last partial page
 770      BEQ finish1
 780 .loop12
 790      LDA(SOURCE),Y
 800      STA(DEST),Y
 810      INY                 \ next char
 820      DEX                 \ any chars left?
 830      BNE loop12
 840 .finish1
 850      RTS
 860
 870 \ ---- procedure mc3 ----
 880 \ list chunk of memory to screen
 890
 900      LDY #0
 910      LDX NUMBER+1        \ first blocks
 920      BEQ restp           \ of 256 chars
 930 .loopp1
 940      LDA (START),Y
 950      JSR OSASCI
 960      INY                 \ next char
 970      BNE loopp1          \ block finished?
 980      INC START+1         \ next block
 990      DEX                 \ any blocks left?
1000      BNE loopp1
```

```
1010 .restp
1020      LDX NUMBER      \ last partial block
1030      BEQ finishp
1040 .loopp2
1050      LDA(START),Y
1060      JSR OSASCI
1070      INY             \ next char
1080      DEX             \ finished?
1090      BNE loopp2
1100 .finishp
1110      RTS
1120
1130 \ ---- procedure mc4 ----
1140 \ list number of spaces to screen.
1150 \ Used to blank out rest of screen
1160 \ at start of insert mode, and to
1170 \ clear the rest of a line.
1180
1190      LDA #32       \ ascii code for space
1200      LDY #0
1210      LDX NUMBER+1 \ blocks of 256 spaces
1220      BEQ rests
1230 .loops1
1240      JSR OSWRCH    \ write to screen
1250      INY           \ block finished?
1260      BNE loops1
1270      DEX           \ any more blocks?
```

```
1280      BNE loops1
1290 .rests
1300      LDX NUMBER    \ last partial block
1310      BEQ finishs
1320 .loops2
1330      JSR OSWRCH
1340      INY
1350      DEX           \ finished?
1360      BNE loops2
1370 .finishs
1380      RTS
1390
1400 \ ---- function mc5 ----
1410 \ move back 'number' lines in memory
1420
1430      LDX NUMBER      \ number of lines
1440      INX             \ number of c.r. 's
1450      LDY #0
1460 .dloop
1470      JSR dstart      \ decrement start addr.
1480      LDA (START),Y
1490      CMP #&FF        \ start of file?
1500      BEQ dexit
1510      CMP #13         \ carriage return?
1520      BNE dloop
1530      DEX             \ next line
1540      BNE dloop       \ finished?
```

```
1550 .dexit
1560      INC START       \ start of next line
1570      BNE dfin
1580      INC START+1     \ for page boundary
1590 .dfin
1600      RTS
1610 .dstart
1620      LDA START
1630      BNE dnonz       \ crossing page boundary?
1640      LDA #&FF
1650      STA START
1660      DEC START+1
1670      BNE dfinish
1680 .dnonz
1690      DEC START       \ no, so just decrement
1700 .dfinish
1710      RTS
1720
1730 \ ---- procedure MC6 ----
1740 \ setup 'x' array for current page
1750 \ used with 'next' and 'previous' commands
1760
1770       LDA #0         \ initial value
1780       LDY #24        \ length of array
1790 .init
1800       STA (DEST),Y   \ first clear array
1810       DEY
```

```
1820        BNE init        \ done?
1830        STA (DEST),Y    \ array(0)
1840        LDX #0          \ array index
1850 .nloop
1860        JSR istart      \ increment start addr.
1870        INC NUMBER      \ current line length
1880        LDA (START),Y
1890        CMP #&FF        \ end of file?
1900        BEQ nexit
1910        CMP #13         \ end of line?
1920        BNE nloop
1930        JSR nfill       \ array(X) = number
1940        CPX #25         \ end of page?
1950        BNE nloop
1960 .nexit
1970        RTS
1980 .istart
1990        INC START
2000        BNE nexit       \ page boundary crossed?
2010        INC START+1
2020        RTS
2030 .nfill
2040        TXA             \ move X to Y
2050        TAY
2060        LDA NUMBER
2070        STA (DEST),Y    \ array(Y) = number
2080        LDA #0
```

```
2090        STA NUMBER      \ for next line
2100        TYA             \ restore X
2110        TAX
2120        INX             \ next line
2130        LDY #0          \ restore Y
2140        RTS
2150 ]
2160  NEXT
2170  PAGE=&6800
2180  CHAIN"EDIT2"
```



```
 1 REM! editor main program
 2 REM! ----------------------------
 3 REM! A.F. de Geus, J.H. Jongejan
 4 REM! and A.M. Koelmans 1984
 5 REM! ----------------------------
 6 REM! Program subject to copyright
 7 REM! ----------------------------
10 ON ERROR GOTO 330
20 VDU23;10;0;0;0;
30 *FX4,1
40 Z%=PAGE-S%-1 : DIM X 24,Y%(25) : Y%(0)=0
50 SOURCE=&70 : DEST=&72 : NUMBER=&74 : START=&70
60 PROCemsg : PROChome : INS%=0
```

```
 70 REPEAT : KEY%=GET
 80   IF (KEY%>=136)AND(KEY%<=139) THEN PROCarrow
 90   IF KEY%=127 THEN PROCdel
100   IF KEY%=1 THEN PROC1mode
110   IF KEY%=14 THEN PROCnext(FALSE)
120   IF KEY%=15 THEN PROCnext(TRUE)
130   IF KEY%=16 THEN PROCprev(FALSE)
140   IF KEY%=17 THEN PROCprev(TRUE)
150   IF KEY%=12 THEN PROClast
160   IF KEY%=13 THEN PROCret
170   IF KEY%=11 THEN PROCkillrol
180   IF KEY%=21 THEN PROCkill
190   IF KEY%=8 THEN PROChome
200   IF KEY%=6 THEN PROCfirst
210   IF KEY%=7 THEN PROClfil
220   IF KEY%=19 THEN PROCsfil
230   IF KEY%=26 THEN PROCzfil
240   IF KEY%=10 THEN PROCjoin
250   IF (KEY%<32)OR(KEY%>126) THEN 280
260   IF C%>=X?L%-1 THEN  VDU7 : GOTO280
270   ?FNmempos=KEY% : VDU KEY% : C%=C%+1
280   UNTILKEY%=3
290 MODE7 : *FX4,0
300 PAGE=&1900 : ?PAGE=13 : ?(PAGE+1)=&FF
310 END
320
330 MODE7
```

```
340 REPORT : PRINT" at ";ERL
350 *FX4,0
360 END
370
380 DEF FNmempos=U%+Y%(L%)+C%
390
400 DEF PROCemsg
410 PRINTCHR$(130)"**END**";
420 ENDPROC
430
440 DEF PROCsetup
450 PROCmc6(U%-1,X)
460 FORI%=1TO25 : Y%(I%)=Y%(I%-1)+X?(I%-1) : NEXT
470 ENDPROC
480
490 DEF PROCshow
500 CLS : PROCsetup
510 IF Y%(25)=0 THEN  PROCemsg : PROChome : ENDPROC
520 PROCmc3(U%,Y%(25)-1)
530 IF X?24=0 THEN PRINT : PROCemsg
540 PRINTTAB(C%,L%);
550 ENDPROC
560
570 DEF PROCfirst
580 L%=0 : C%=0 : U%=S% : PROCshow
590 ENDPROC
600
```

```
610 DEF PROCnext(HALF%)
620 C%=0 : L%=0
630 IF HALF% THEN U%=U%+Y%(12) ELSE U%=U%+Y%(25)
640 PROCshow
650 ENDPROC
660
670 DEF PROChome
680 C%=0 : L%=0 : PRINTTAB(0,0);
690 ENDPROC
700
710 DEF PROCret
720 VDU13 : C%=0
730 ENDPROC
740
750 DEF PROCarrow
760 ON (KEY%-135)GOTO770,790,810,840
770 IF C%=0 THEN  ENDPROC
780 C%=C%-1 : VDU8 : ENDPROC
790 IF C%=39 THEN  ENDPROC
800 C%=C%+1 : VDU9 : ENDPROC
810 IF L%=24 THEN ENDPROC
820 IF X?(L%+1)=0 THEN ENDPROC
830 L%=L%+1 : VDU10 : ENDPROC
840 IF L%=0 THEN  ENDPROC
850 L%=L%-1 : VDU11 : ENDPROC
860
870 DEF PROC1mode
```

```
 880 IF (X?0>0)AND(C%>=X?L%) THEN VDU7 : ENDPROC
 890 M%=FNmempos : N%=E%-M%+1 : R%=LEN($M%)+1 : INS%=0
 900 PROCmc1(M%,PAGE-N%,N%)
 910 PROCmc4(1000-L%*40-C%-1) : PRINTTAB(C%,L%);
 920 REPEAT : KEY%=GET
 930   IF (KEY%<>13)AND((KEY%<32)OR(KEY%>127)) THEN 1000
 940   IF KEY%=127  THEN 980
 950   M%?INS%=KEY% : C%=C%+1 : INS%=INS%+1 : Z%=Z%-1
 960   IF KEY%=13 THEN PROCimret
 970   IF C%>38 THEN PROCins
 980   IF KEY%=127  THEN PROCidel
 990   IF (KEY%<>13)AND(KEY%<>127) THEN  VDUKEY%
1000 UNTIL KEY%=1
1010 IF C%+R%>=40 THEN VDU7 : GOTO920
1020 PROCmc2(PAGE-N%,M%+INS%,N%)
1030 E%=E%+INS% : E%?1=&FF
1040 IF ?E%=&FF  THEN E%=E%-1
1050 PROCshow
1060 ENDPROC
1070
1080 DEF PROCins
1090 LOCAL IN% : IN%=INS%
1100 IF M%?(IN%-1)=32 THEN M%?(IN%-1)=13 : PROCimret : VDU127 :
ENDPROC
1110 REPEAT : IN%=IN%-1 : UNTIL(IN%=0 OR M%?IN%=32 OR M%?IN%=13)
1120 IF (IN%=0 OR M%?IN%=13) THEN  PROCinsret : ENDPROC
1130 M%?IN%=13 : L%=L%+1 : C%=INS%-IN%-1
```

```
1140 IF Z%<100 THEN SOUND1,-15,200,20
1150 IF L%>24  THEN PROCimnew : VDU127 : ENDPROC
1160 K%=INS%-IN%-1 : FORI%=1 TO K% : VDU127 : NEXT : C%=K%
1170 VDU13 : VDU10 : FORI%=0 TO K%-1 : VDU(M%?(IN%+I%)) : NEXT
1180 ENDPROC
1190
1200 DEF PROCimret
1210 IF Z%<100 THEN SOUND1,-15,200,20
1220 L%=L%+1 : C%=0
1230 IF L%>24 THEN PROCimnew ELSEVDU13 : VDU10
1240 ENDPROC
1245
1250 DEF PROCinsret
1260 M%?INS%=13 : INS%=INS%+1 : Z%=Z%-1
1270 VDUKEY% : KEY%=13 : PROCimret
1280 ENDPROC
1285
1290 DEF PROCidel
1300 IF (C%=0)OR(INS%=0) THEN VDU7 : ENDPROC
1310 C%=C%-1 : INS%=INS%-1 : Z%=Z%+1 : VDU127
1320 ENDPROC
1325
1330 DEF PROCimnew
1340 LOCALNN%
1350 L%=10 : NN%=M%+INS% : CLS
1360 U%=FNmc5(NN%,10) : PROCmc3(U%,NN%-U%)
1370 ENDPROC
```

```
1375
1380 DEF PROCmc1(AA%,BB%,CC%)
1390 !SOURCE=AA% : !DEST=BB% : !NUMBER=CC%
1400 CALL&C00
1410 ENDPROC
1420
1430 DEF PROCmc2(AA%,BB%,CC%)
1440 !SOURCE=AA% : !DEST=BB% : !NUMBER=CC%
1450 CALL&C30
1460 ENDPROC
1470
1480 DEF PROCmc3(AA%,BB%)
1490 !START=AA% : !NUMBER=BB% : CALL&C51
1500 ENDPROC
1510
1520 DEF PROCmc4(AA%)
1530 !NUMBER=AA% : CALL&C72
1540 ENDPROC
1550
1560 DEF FNmc5(AA%,BB%)
1570 !START=AA% : !NUMBER=BB% : CALL&C8F
1580 =!START AND &FFFF
1590
1600 DEF PROCmc6(AA%,BB%)
1610 !START=AA% : !DEST=BB% : !NUMBER=0
1620 CALL&CBA
1630 ENDPROC
```

```
1640
1650 DEF PROCwipe(NN%)
1660 M%=FNmempos
1670 PROCmc2(M%+NN%,M%,E%-M%+2)
1680 X?L%=X?L%-NN%
1690 FORI%=L%+1TO25 : Y%(I%)=Y%(I%)-NN% : NEXT
1700 E%=E%-NN% : Z%=Z%+NN%
1710 ENDPROC
1720
1730 DEF PROCdel
1740 IF C%>=X?L%-1 THEN VDU7 : ENDPROC
1750 PROCwipe(1)
1760 PROCmc3(M%,X?L%-C%-1) : VDU32
1770 PRINTTAB(C%,L%);
1780 ENDPROC
1790
1800 DEF PROCkillrol
1810 IF C%>=X?L%-1 THEN VDU7 : ENDPROC
1820 N%=X?L%-C%-1 : PROCwipe(N%)
1830 PROCmc4(N%) : PRINTTAB(C%,L%);
1840 ENDPROC
1850
1860 DEF PROCjoin
1870 M%=FNmempos
1880 IF (?M%<>13)OR(L%=24) THEN VDU7 : ENDPROC
1890 IF X?L%+X?(L%+1)>39 THEN VDU7 : ENDPROC
1900 PROCmc2(M%+1,M%,E%-M%+2)
```

```
1910 E%=E%-1 : PROCshow
1920 ENDPROC
1930
1940 DEF PROCkill
1950 C%=0 : PROCwipe(X?L%)
1960 PROCshow
1970 ENDPROC
1980
1990 DEF PROCprev(HALF%)
2000 IF HALF% THEN U%=FNmc5(U%,12)
2010 IF NOT HALF% THEN U%=FNmc5(U%,25)
2020 PROCshow
2030 ENDPROC
2040
2050 DEF PROClast
2060 U%=FNmc5(E%+1,25) : PROCshow
2070 ENDPROC
2075
2080 DEF PROCzfil
2090 C%=0 : CLS
2100 PRINT : INPUT"LOAD WHICH FILE? "F$
2110 CH%=OPENIN(F$)
2120 IF EXT#(CH%)>Z%-10  THEN PRINTCHR$(129)"FILE TOO LONG" :
GOTO350
2130 REPEAT : C%=C%+1
2140  IF C%>39 THEN ?E%=&0D : E%=E%+1 : C%=0
2150  A%=BGET£CH%
2160  ?E%=A% : IF A%=&0D  THEN C%=0
```

```
2170  E%=E%+1
2180  UNTILEOF£CH%
2190 CLOSE£CH%
2200 ?E%=&FF : E%=E%-1 : Z%=PAGE-E%-1
2210 PROCfirst
2220 ENDPROC
2225
2230 DEF PROCsfil
2240 CLS
2250 PRINT : INPUT"SAVE TO WHICH FILE? "F$
2260 PROCcli("*SAVE "+F$+" "+STR$~(S%)+" "+STR$~(E%+1))
2270 PROCshow
2280 ENDPROC
2285
2290 DEF PROClfil
2300 CLS : C%=0
2310 PRINT : INPUT"*LOAD WHICH FILE? "F$
2320 CH%=OPENIN(F$) : E%=S%+EXT#(CH%)
2330 CLOSE#CH% : ?E%=&FF : E%=E%-1 : Z%=PAGE-E%-1
2340 PROCcli("*LOAD "+F$+" "+STR$~(S%))
2350 PROCfirst
2360 ENDPROC
2370
2380 DEF PROCcli(S$)
2390 DIM C 40
2400 $C=S$
2410 X%=C MOD256 : Y%=C DIV256 : CALL&FFF7
2420 ENDPROC
```

APPENDIX F

The scanner listing

Notes on the program

10 initialise the program, open files etc.

200 finite state machine to determine character frequencies on string file. There are four states:

1 count characters
2 end of line detected
3 # found on first position of line
4 $ found on first position of line

The variable more% tells whether or not a new character needs to be read from the input file

380 initialise the list of characters and their frequencies

410 build the Huffman Code

460 store the codes and their frequencies from &900 onwards. A maximum of 24 bits per code is assumed.

500 this is essentially the same finite state machine as before, but instead of counting the characters their codes are written to file, and the start addresses of all messages are saved

640 write the start addresses to file

670 write the Huffman tree to file

720 declare arrays used to store information from the other input files and initialise them

800 read the dictionary

820 write the hash table to file

930 read the transition table

940 declare and initialise the variables needed to store information from the syntax file

990 read the syntax file

function readnum
reads a number after a £ has been found on the first position of the line

procedure schar
procedure slink
procedure sleft
procedure sright
these procedures set various fields of a tree node. A tree node is a cluster of four bytes treated as a single entity

function gchar
function glink
function gleft
function gright
these functions retrieve the value of a field in a tree node

function newtree
allocates a new tree node

function newchar
allocates a new character value for an internal tree node

function strtoint
converts a string consisting of zeros and ones to an integer (the string is read backwards to facilitate encoding)

procedure insert
inserts a new tree node into the current list. There are four cases to be dealt with:
–the list is empty
–it must be inserted at the head of the list
–it must be inserted at the tail of the list
–it must be inserted somewhere in the middle

procedure encode
encodes the characters in the string file using the Huffman Code. It is written in machine code because it would take BASIC ages to do it.

1530 declare the keywords of the ADL, the predefined procedures and functions, the object classes, the predefined variables, and (part of) the various error messages

procedure sh1
sets id field in user hash table

procedure sh2 sets
    link field in user hash table

procedure sr1
    sets id field in reserved words hash table

procedure sr2
    sets link field in reserved words hash table

function gh1
function gh2
function gr1
function gr2
    these functions retrieve the corresponding values

procedure slu
    keeps track of which locations are being used

function hash
    the hash function

function lookup
    lookup a word in one of the hash tables

function enter
    enter a word in one of the hash tables

procedure p
    indicate that the last character read from the current input file is still valid

function g
    get the next character from the current input file

procedure slin
    skip to the end of the line

function sspa
    skip spaces until a non-space or end-of-file is found

function rword
    read a word or number from the input file

procedure err
    print general error messages

procedure derr
    print dictionary error messages

procedure terr
    print transition table error messages

function ritem
    read a word (item) with an expected type from the input file. The type of the word can in most cases be deduced from the id field in the hash table

function dict
    read the dictionary

procedure rtrans
    read the transition table

procedure serr
    print syntax file error messages

function gitem
    get the next item from the syntax file

function idnum
    get the id value of a keyword from the reserved words hash table

procedure sitem
    skip items until the specified item is found. This procedure is called after an error has been found

procedure out
    put a byte in the syntax output buffer

procedure pverb
    write the output buffer to file

procedure syntax
    read the syntax file

function stat
    read a statement from the syntax file

function ifstat
    read an if-statement from the syntax file

function cond
    read a condition from the syntax file

function cstat
    read a case-statement from the syntax file

function csel
read a case-selector from the syntax file

function action
read an action from the syntax file

procedure cl
reset 'vartop' (a BASIC pseudo variable) to the current value of V% and unlink all variables above V% from the BASIC variable table. This rather disgusting trick is unfortunately necessary because of the Beeb's rather limited memory...

procedure tc
save the current output file pointer to the second output file

procedure outc
write a byte to the output file

procedure tc2
write a constant byte to the second output file

```
    1 REM! Scanner for Adventure Generator
    2 REM! ------------------------------
    3 REM! A.F. de Geus, J.H. Jongejan
    4 REM! and A.M. Koelmans 1985
    5 REM! ------------------------------
    6 REM! Program subject to copyright
    7 REM! To run, type "chain scanner"
    8 REM! ------------------------------
   10CLIM%=101:ALIM%=151:RLIM%=201:TLIM%=128
   20VAR%=1:CLASS%=2:COND%=3:ACT%=4:RESW%=5
   30MVERB%=6:NVERB%=7:TVERB%=8:OBJ%=9:MESNR%=10
   40LOC%=11:OBJSTR%=12:ID%=13:VVAL%=14:PVAL%=15
   50NOITEM%=64:VERB%=128:NNOTYP%=256:NDEFV%=5
   60NTRANS%=5:WLEN%=8:MAXLOC%=255
   70CEOF%=-1:CEOL%=-2:CERR%=-3:CNOR%=-4:CRES%=-5
   80w$=STRING$(25," "):L%=1:f$="$":@%=4:DIMmax 2:!max=0
   90chan%=0:vchan%=0:head%=0:tail%=0:tchan%=0:V%=!2AND&FF
FF
  100CLS:REPEAT:INPUT"DIR:"f$:UNTILLEN(f$)=1
  110H%=ASC("#"):D%=ASC("$"):more%=TRUE
  120vchan%=OPENOUT(f$+".1"):tchan%=OPENOUT(f$+".2"):GOTO7
20
  130PROCcl:state%=2:sec%=0:more%=TRUE
  140DIMcode 250,chars%(255),tree 756,code$(126),addr%(2,2
55)
  150base1=&70:base2=&72:membuf=&74:cnt=&75:infile=&76
  160outfile=&77:temp=&78:flag=&79:mempt=&7A
```

```
  170osbput=&FFD4:osbget=&FFD7:PROCencode
  180chan%=OPENIN(f$+".STRING"):PRINT"Strings";
  190IFchan%=0PRINT"?":END:ELSEPRINT
  200REPEAT:IFmore%C%=BGET#chan%
  210IFEOF#chan%UNTILTRUE:GOTO360
  220IFsec%>2THEN360
  230ONstate%GOSUB240,270,300,330:UNTILFALSE
  240chars%(C%)=chars%(C%)+1:more%=TRUE
  250IFC%<>13REPEAT:C%=BGET#chan%:chars%(C%)=chars%(C%)+1:
UNTILC%=13
  260state%=2:RETURN
  270IFC%=H%state%=3:more%=FALSE:RETURN
  280IFC%=D%state%=4:RETURN
  290more%=FALSE:state%=1:RETURN
  300chars%(10)=chars%(10)+1:N%=FNreadnum:state%=1
  310IFN%>max?sec%max?sec%=N%
  320RETURN
  330sec%=sec%+1:IFsec%>2RETURN
  340IFC%<>13REPEAT:C%=BGET#chan%:UNTILC%=13
  350state%=2:RETURN
  360IFsec%<2PRINT"String sections?"
  370PTR#chan%=0:PRINT"Hufman code":T%=0:C%=127
  380FORI%=10TO254:IFchars%(I%)=0THEN400
  390J%=FNnewtree:PROCschar(J%,I%):PROCinsert(J%)
  400NEXT
  410REPEAT:second%=FNglink(head%):third%=FNglink(second%)
  420J%=FNnewtree:PROCschar(J%,FNnewchar)
```

```
  430chars%(FNgchar(J%))=chars%(FNgchar(head%))+chars%(FNg
char(second%))
  440PROCsleft(J%,head%):PROCsright(J%,second%):head%=thir
d%:PROCinsert(J%)
  450UNTILthird%=0:PROCinfo(head%,"")
  460FORI%=10TO126:I%?&900=LEN(code$(I%))
  470K%=FNstrtoint(code$(I%)):I%?&A00=K%MOD256:K%=K%DIV256
  480I%?&B00=K%MOD256:I%?&C00=K%DIV256:NEXT:PROCtc
  490?&76=chan%:?&77=vchan%:state%=2:sec%=0:more%=TRUE:N%=
0:PRINT"Encoding"
  500REPEAT:IFmore%C%=BGET#chan%
  510IFEOF#chan%UNTILTRUE:GOTO630
  520IFsec%>2THEN630
  5300Nstate%GOSUB540,550,590,600:UNTILFALSE
  540CALLencode:C%=?&79:more%=FALSE:state%=2:RETURN
  550IFC%=H%state%=3:more%=FALSE:RETURN
  560IFC%=D%state%=4:RETURN
  570IFNOTEOF#chan%more%=TRUE
  580RETURN
  590N%=FNreadnum:addr%(sec%,N%)=!mempt:state%=1:RETURN
  600sec%=sec%+1:more%=TRUE:N%=0:IFsec%>2RETURN
  610IFC%<>13REPEAT:C%=BGET#chan%:UNTILC%=13
  620state%=2:RETURN
  630CLOSE#chan%
  640FORS%=0TO2:PROCtc:FORN%=0TOmax?S%
  650PROCoutc(addr%(S%,N%)AND&FF):PROCoutc((addr%(S%,N%)DI
V256)AND&FF)
```

```
  660NEXTN%:NEXTS%:Z%=&1300+PTR#vchan%
  670W%=&900:E%=FNhuftree(head%)
  680PROCtc:FORE%=&900TOW%:PROCoutc(?E%):NEXT
  690PROCtc2(&E5B):PROCtc2(&D00):PROCtc2(&E8E):PROCtc2(&EA
E)
  700IFPTR#vchan%>&6900PRINTCHR$129,"Game too large!"
  710CLOSE#0:END
  720DIMhstr$(250),rstr$(250),er$(12),terr$(5)
  730hash%=&900:resvw%=&B00:DIMobjects%320,loctab%2550
  740FORI%=0TO250:PROCsh2(I%,-1):PROCsr2(I%,-1):NEXT
  750FORI%=0TO250:PROCsh1(I%,-1):PROCsr1(I%,-1):NEXT
  760FORI%=1TO54:READw$,J%:N%=FNenter(w$,J%,FALSE):NEXT
  770FORI%=0TONDEFV%:READw$:N%=FNenter(w$,I%,FALSE):NEXT
  780FORI%=1TO12:READer$(I%):NEXT:FORI%=1TO5:READterr$(I%)
:NEXT
  790chan%=OPENIN(f$+".DICT"):IFchan%=0PRINT("dict?"):END
  800PRINT"Dict":N%=FNdict:IFN%<>0PRINT"only",N%,"sections
 in dict":END
  810PROCtc:i%=-1:cnt%=0
  820REPEAT:i%=i%+1:IFi%>250PROCoutc(&FE):UNTILTRUE:GOTO89
0
  830IFhstr$(i%)=""cnt%=cnt%+1:GOTO880
  840IFcnt%=1PROCoutc(0):GOTO860
  850IFcnt%>1PROCoutc(&FF):PROCoutc(cnt%)
  860cnt%=LEN(hstr$(i%)):IFcnt%>6cnt%=6
  870PROCoutc(cnt%):FORj%=1TOcnt%:PROCoutc(ASC(MID$(hstr$(
i%),j%,1))):NEXT:cnt%=0
```

```
  880UNTILFALSE
  890PROCtc:FORi%=0TO250:cnt%=FNgh1(i%):IFcnt%<>&FFPROCout
c(cnt%)
  900NEXT
  910CLOSE#chan%:DIMlu%255
  920chan%=OPENIN(f$+".TRANS"):IFchan%=0PRINT"trans?":END
  930PRINT"Trans":L%=1:PROCrtrans:CLOSE#chan%
  940DIMsact%80,outbuf%511,vtab%(191):N%=0:O%=0:L%=1:F%=0
  950FORI%=0TO26:FORJ%=0TO1
  960READw$:sact%?(3*I%+J%)=EVAL(w$+"%"):NEXTJ%:sact%?(3*I
%+2)=0:NEXTI%
  970FORI%=0TO191:vtab%(I%)=-1:NEXT
  980chan%=OPENUP(f$+".SYNTAX"):IFchan%=0PRINT"syntax?":EN
D
  990PRINT"Syntax":PROCsyntax:CLOSE#chan%:GOTO130
 1000DEFFNreadnum:LOCALn%:n%=0:REPEAT:C%=BGET#chan%
 1010IFC%>=48ANDC%<=57n%=10*n%+C%-48
 1020UNTILC%<48ORC%>57:IFn%>255PRINT"Line num? at line",ch
ars%(13)+1
 1030=n%
 1040DEFPROCschar(i%,v%):tree?(4*i%)=v%:ENDPROC
 1050DEFPROCslink(i%,v%):tree?(4*i%+1)=v%:ENDPROC
 1060DEFPROCsleft(i%,v%):tree?(4*i%+2)=v%:ENDPROC
 1070DEFPROCsright(i%,v%):tree?(4*i%+3)=v%:ENDPROC
 1080DEFFNgchar(i%)=tree?(4*i%)
 1090DEFFNglink(i%)=tree?(4*i%+1)
 1100DEFFNgleft(i%)=tree?(4*i%+2)
```

```
 1110DEFFNgright(i%)=tree?(4*i%+3)
 1120DEFFNnewtree:T%=T%+1:PROCslink(T%,0):=T%
 1130DEFFNnewchar:C%=C%+1:=C%
 1140DEFFNstrtoint(c$):LOCALl%:l%=0:IFLEN(c$)=0THEN=0
 1150FORJ%=LEN(c$)TO1STEP-1:l%=2*l%+ASC(MID$(c$,J%,1))-ASC
("0"):NEXT:=l%
 1160DEFPROCinsert(l%):LOCALll%,oo%:IFhead%=0head%=l%:tail
%=l%:ENDPROC
 1170IFchars%(FNgchar(l%))<=chars%(FNgchar(head%))PROCslin
k(l%,head%):head%=l%:ENDPROC
 1180IFchars%(FNgchar(l%))>=chars%(FNgchar(tail%))PROCslin
k(tail%,l%):tail%=l%:ENDPROC
 119011%=head%:REPEAT:oo%=11%:11%=FNglink(11%)
 1200UNTILchars%(FNgchar(l%))<=chars%(FNgchar(11%))
 1210PROCslink(oo%,l%):PROCslink(l%,11%):ENDPROC
 1220DEFPROCinfo(b%,c$):LOCALbb%,cc$:bb%=FNgchar(b%)
 1230IFbb%<128code$(bb%)=c$:ENDPROC
 1240cc$=c$+"0":PROCinfo(FNgleft(b%),cc$)
 1250cc$=c$+"1":PROCinfo(FNgright(b%),cc$):ENDPROC
 1260DEFFNhuftree(root%):LOCALbb%,cc%,dd%,zz%:bb%=W%:cc%=F
Ngchar(root%):zz%=Z%
 1270IFcc%<128THEN?W%=cc%:Z%=Z%+1:W%=W%+1:=zz%
 1280?W%=0:W%=W%+5:Z%=Z%+5:dd%=FNhuftree(FNgleft(root%))
 1290bb%?1=dd%MOD256:bb%?2=dd%DIV256:dd%=FNhuftree(FNgrigh
t(root%))
 1300bb%?3=dd%MOD256:bb%?4=dd%DIV256:=zz%
 1310DEFPROCencode:!mempt=0:FORI%=0TO2STEP2:P%=code:[OPTI%
```

```
 1320.encode:LDY#0:STYmembuf:STYcnt:STYbase1:STYbase2:STYf
lag
 1330LDY#9:STYbase1+1:.gchar:LDY#&0A:STYbase2+1:LDYinfile:
JSRosbget:BCC10
 1340JMPsexit:.10:CMP#13:BNE11:JSRsflg:JMP12
 1350.11:LDYflag:BEQ12:JSRcflg:CMP£ASC("#"):BNE13:STAflag:
JMPsexit
 1360.13:CMP#ASC("$"):BNE12:STAflag:JMPsexit
 1370.12:TAY:LDA(base1),Y:BEQgchar:JSRenc:JMPgchar
 1380.sexit:LDY#10:LDA(base1),Y:JSRenc:LDXcnt:BEQfinish
 1390.floop:CLC:ROLmembuf:INX:CPX#8:BNEfloop
 1400LDAmembuf:LDYoutfile:JSRosbput:JSRimempt:.finish:RTS
 1410.enc:TAX:.loop:LDA(base2),Y:CPX#9:BCCsmthan8:JSRbyteo
ne
 1420TXA:SEC:SBC#8:TAX:INCbase2+1:JMPloop
 1430.smthan8:LSRA:JSRsetbit:DEX:BNEsmthan8:RTS
 1440.setbit:PHA:TXA:PHA:LDXcnt:ROLmembuf:INX:CPX#8:BEQout
c:STXcnt:BNEexit
 1450.outc:TYA:PHA:LDAmembuf:LDYoutfile:JSRosbput:JSRimemp
t
 1460PLA:TAY:LDA#0:STAcnt:STAmembuf:.exit:PLA:TAX:PLA:RTS
 1470.byteone:STAtemp:TXA:PHA:LDX#8:LDAtemp
 1480.obloop:LSRA:JSRsetbit:DEX:BNEobloop:PLA:TAX:RTS
 1490.imempt:INCmempt:BNEiret:INCmempt+1:.iret:RTS
 1500.sflg:PHA:LDA#1:STAflag:PLA:RTS
 1510.cflg:PHA:LDA#0:STAflag:PLA:RTS:]:NEXT:ENDPROC
 1520]:NEXT:ENDPROC
```

```
 1530DATA"IF",201,"THEN",202,ELIF,203,"ELSE",204,FI,205
 1540DATACASE,206,OF,207,DEFAULT,208,ESAC,209,"NEXT",210
 1550DATA"END",211,"NOT",212,"AND",213,"OR",214,"EOF",215
 1560DATA",",216,"?",217,":",218,INVENT,230,"LOCAL",231
 1570DATASKIP,232,CHLOC,151,CHCLASS,152,CHOSTR,153
 1580DATASETRAN,154,SETVAR,155,ADD,156,SUB,157,"PRINT",158
 1590DATA"READI",159,REPROC,160,"LISTLOC",161,MESSAGE,162,
SLEEP,163
 1600DATA"PRINTA",164,QUIT,165,CHLSTR,166,ALARM,167,PRESEN
T,101
 1610DATAHOLDING,102,TESTCL,103,EQUAL,104,SMALLER,105,SMEQ
UAL,106
 1620DATA"READB",107,OBJSTR,108,LASTOBJ,109,OBJLOC,110,FRI
END,51
 1630DATAENEMY,52,TREASUR,53,AID,54,SPECIAL,55,NOCLASS,56
 1640DATACURLOC,"OLDLOC",SCORE,CUROBJ,NTURNS,DUMMY
 1650DATAvariable,object class,condition,action,reserved w
ord,move verb
 1660DATAnon-transitive verb,transitive verb,object
 1670DATAmessage number,location,identification
 1680DATAmissing transition,too many transitions,no transi
tions
 1690DATAis isolated,has no further transitions
 1700DATAOBJ,LOC,OBJ,CLASS,OBJ,OBJSTR,VAR,PVAL,VAR,VVAL,VA
R,VVAL
 1710DATAVAR,VVAL,VAR,N,VAR,N,VERB,OBJ,LOC,N,MESNR,N,VVAL,
N,VVAL,VVAL
```

```
 1720DATAN,N,LOC,LOC,VERB,VVAL,OBJ,N,OBJ,N,OBJ,CLASS,VAR,V
VAL
 1730DATAVAR,VVAL,VAR,VVAL,N,N,OBJ,OBJSTR,OBJ,N,OBJ,LOC
 1740DEFPROCsh1(I%,J%):hash%?(2*I%)=J%:ENDPROC
 1750DEFPROCsh2(I%,J%):hash%?(2*I%+1)=J%:ENDPROC
 1760DEFPROCsr1(I%,J%):resvw%?(2*I%)=J%:ENDPROC
 1770DEFPROCsr2(I%,J%):resvw%?(2*I%+1)=J%:ENDPROC
 1780DEFFNgh1(I%)=hash%?(2*I%)
 1790DEFFNgh2(I%)=hash%?(2*I%+1)
 1800DEFFNgr1(I%)=resvw%?(2*I%)
 1810DEFFNgr2(I%)=resvw%?(2*I%+1)
 1820DEFPROCslu(I%,J%):lu%?I%=(lu%?I%)ORJ%:ENDPROC
 1830DEFFNhash(s$):LOCALi%,j%,v%:v%=0
 1840FORi%=1TOLEN(s$):j%=ASC(MID$(s$,i%,1)):v%=(v%*19+j%)M
OD251:NEXT:=v%
 1850DEFFNlookup(s$,flag%):LOCALi%:i%=FNhash(s$)
 1860IFNOTflag%THEN1900
 1870REPEAT:IFs$=hstr$(i%)UNTILTRUE:=i%
 1880IFFNgh2(i%)=&FFUNTILTRUE:=&FF
 1890i%=FNgh2(i%):UNTILFALSE
 1900REPEAT:IFs$=rstr$(i%)UNTILTRUE:=i%
 1910IFFNgr2(i%)=&FFUNTILTRUE:=&FF
 1920i%=FNgr2(i%):UNTILFALSE
 1930DEFFNenter(s$,id%,flag%)
 1940LOCALi%,s%:i%=FNhash(s$):IFNOTflag%THEN2020
 1950IFFNgh2(i%)<>&FFREPEAT:i%=FNgh2(i%):UNTILFNgh2(i%)=&F
F
```

```
 1960s%=i%:IFhstr$(i%)=""THEN2000
 1970REPEAT:i%=i%+1:IFi%>250i%=0
 1980IFi%=s%PRINT"hashtab?":END
 1990UNTILhstr$(i%)=""
 2000IFs%<>i%PROCsh2(s%,i%)
 2010hstr$(i%)=s$:PROCsh1(i%,id%):=i%
 2020IFFNgr2(i%)<>&FFREPEAT:i%=FNgr2(i%):UNTILFNgr2(i%)=&F
F
 2030s%=i%:IFrstr$(i%)=""THEN2070
 2040REPEAT:i%=i%+1:IFi%>250i%=0
 2050IFi%=s%PRINT"hashtab?":END
 2060UNTILrstr$(i%)=""
 2070IFs%<>i%PROCsr2(s%,i%)
 2080rstr$(i%)=s$:PROCsr1(i%,id%):=i%
 2090DEFPROCp:more%=FALSE:ENDPROC
 2100DEFFNg:IFNOTmore%more%=TRUE:=C%
 2110C%=BGET#chan%:IFC%=13L%=L%+1
 2120=C%
 2130DEFPROCslin:LOCALc%
 2140REPEAT:IFEOF#chan%UNTILTRUE:ENDPROC
 2150c%=FNg:IFc%=13UNTILTRUE:ENDPROC
 2160UNTILFALSE
 2170DEFFNsspa:LOCALc%
 2180REPEAT:IFEOF#chan%UNTILTRUE:=CEOF%
 2190c%=FNg:IFc%=13PROCp:UNTILTRUE:=CEOL%
 2200IFc%<>32PROCp:UNTILTRUE:=0
 2210UNTILFALSE
```

```
 2220DEFFNrword:LOCALc%,n%:n%=0
 2230w$="":c%=FNsspa:IFc%<0THEN=c%
 2240REPEAT:IFEOF#chan%UNTILTRUE:=CEOF%
 2250c%=FNg:IFc%<=32PROCp:UNTILTRUE:=n%
 2260IF(c%<>44)AND(c%<>58)THEN2280
 2270IFn%=0w$=CHR$(c%):UNTILTRUE:=1:ELSEPROCp:UNTILTRUE:=n
%
 2280IFn%<WLEN%w$=w$+CHR$(c%):n%=n%+1
 2290UNTILFALSE
 2300DEFPROCerr(s$,w$,k%):PRINTs$," in line",L%;
 2310IFw$<>""PRINT", item",w$:ELSEPRINT
 2320IFk%>0ANDk%<13PRINTer$(k%)," expected"
 2330ENDPROC
 2340DEFPROCderr(e%,w$):PRINT"dict: ",w$;
 2350IFe%=1PRINT" already exists";
 2360PRINT" in line",L%:ENDPROC
 2370DEFPROCterr(e%,l1%,w$):PRINT"location",l1%," ";
 2380IFw$<>""PRINT", word ",w$;
 2390IFe%>0ANDe%<6PRINTterr$(e%)ELSEPRINT
 2400ENDPROC
 2410DEFFNritem(typ%):LOCALi%,j%,n%,idd%,k%,wrd$:k%=0
 2420n%=FNrword:wrd$=w$:IFn%>0THEN2460
 2430IF(typ%ANDNOITEM%)THEN=n%
 2440IFn%=CEOF%PROCerr("eof","",k%)ELSEPROCerr("missing it
em","",k%)
 2450=n%
 24601%=typ%ANDNNOTYP%:IFi%=0THEN2490
```

```
2470n%=FNlookup(wrd$,FALSE):IFn%=&FFTHEN=0
2480=FNgr1(n%)
2490k%=typ%MOD64:IF(typ%ANDVERB%)k%=MVERB%
2500IFk%<MVERB%n%=FNlookup(wrd$,FALSE):GOTO2690
2510IFk%<MESNR%n%=FNlookup(wrd$,TRUE):GOTO2640
2520n%=0:IFLEFT$(wrd$,1)<>"?"THEN2570
2530wrd$=MID$(wrd$,2):n%=FNlookup(wrd$,FALSE)
2540IFn%<>&FFn%=FNgr1(n%)
2550IF(n%>=0)AND(n%<=50)THEN=(n%+1000)
2560PROCerr("deref?",wrd$,k%):=CERR%
2570FORi%=1TOLEN(wrd$):j%=ASC(MID$(wrd$,i%,1))
2580IF(j%<48)OR(j%>57)PROCerr("digit?",w$,k%)
2590n%=10*n%+j%-48:NEXT
2600IFn%>=256THEN2630
2610IF(k%=ID%)AND(n%>=64)THEN2630
2620IF(k%=PVAL%)AND(n%>=100)THEN2630ELSE=n%
2630PROCerr("value too big",wrd$,k%):=CERR%
2640IFn%<>&FFTHEN2690
2650IFFNlookup(wrd$,k%<MVERB%)=&FFTHEN2680
2660PROCerr("unexpected item",wrd$,k%)
2670IFk%>=MVERB%THEN=CRES%ELSE=CNOR%
2680PROCerr("unknown item",wrd$,k%)
2690IFk%>=MVERB%THEN2740
2700idd%=(FNgr1(n%)DIV50)+1
2710IF(k%=RESW%)AND(idd%=ACT%)idd%=RESW%
2720IF(k%=COND%)AND(idd%=RESW%)idd%=COND%
2730GOTO2760
```

```
2740idd%=(FNgh1(n%)DIV64)+MVERB%
2750IF(typ%ANDVERB%)AND(idd%<=TVERB%)idd%=MVERB%
2760IFidd%<>k%PROCerr("type?",wrd$,k%):=CERR%
2770IFk%>=MVERB%THEN=FNgh1(n%)ELSE=FNgr1(n%)
2780=CERR%
2790DEFFNdict
2800LOCALi%,n%,sec%,idd%,cl%,tn%,lc%,ar%,wrd$,max%:max%=0
2810FORi%=1TO63:objects%?(5*i%)=&FF:NEXT:sec%=1:i%=NDEFV%
2820IFFNg=D%THEN2880
2830REPEAT:PROCp:n%=FNrword:wrd$=w$:IFn%=CEOF%UNTILTRUE:=
1
2840IFn%<=0THEN2870
2850IFFNlookup(wrd$,FALSE)<>&FFPROCderr(1,wrd$):GOTO2880
2860i%=i%+1:k%=FNenter(wrd$,i%,FALSE)
2870PROCslin:UNTILFNg=D%
2880sec%=2:PROCslin:k%=FNenter("ALLMOV",0,TRUE)
2890REPEAT:IFsec%=5UNTILTRUE:GOTO2970
2900IFFNg=ASC("$")sec%=sec%+1:GOTO2960:ELSEPROCp
2910n%=FNrword:wrd$=w$:IFn%=CEOF%UNTILTRUE:=sec%
2920IFn%<=0THEN2960
2930IFFNlookup(wrd$,TRUE)<>&FFPROCderr(1,wrd$):GOTO2960
2940n%=FNritem(ID%):IFn%=CEOF%UNTILTRUE:=sec%
2950IFn%>0k%=FNenter(wrd$,n%+((sec%-2)*64),TRUE)
2960PROCslin:UNTILFALSE
2970k%=FNenter("CLAY",&FE,TRUE)
2980REPEAT:IFFNg=D%UNTILTRUE:GOTO3150:ELSEPROCp
2990n%=FNrword:wrd$=w$:IFn%<=0THEN3130
```

```
 3000IFwrd$="a"ar%=2:GOTO3010:ELSEIFwrd$="an"ar%=1:ELSEar%
=0
 3010IFar%<>0n%=FNrword:wrd$=w$:IFn%<=0THEN3130
 3020n%=FNlookup(wrd$,TRUE):tn%=n%
 3030IF(n%<>&FF)AND(FNgh1(n%)<192)PROCderr(1,wrd$):GOTO313
0
 3040n%=FNritem(ID%):idd%=n%:IFn%<=0THEN3130
 3050IFidd%>max%max%=idd%
 3060n%=FNritem(CLASS%):c1%=n%:IFn%<=0THEN3130
 3070n%=FNritem(LOC%):lc%=n%:IFn%<0THEN3130
 3080n%=FNritem(OBJSTR%):IFn%<=0THEN3130
 3090IFtn%=&FFtn%=FNenter(wrd$,idd%+192,TRUE)
 3100objects%?(5*idd%)=lc%:objects%?(5*idd%+1)=c1%
 3110objects%?(5*idd%+2)=ar%:objects%?(5*idd%+3)=n%
 3120objects%?(5*idd%+4)=tn%
 3130IFn%=CEOF%UNTILTRUE:=sec%
 3140PROCslin:UNTILFALSE
 3150PROCtc:FORi%=0TOmax%*5+4:PROCoutc(objects%?i%):NEXT
 3160=0
 3170DEFPROCrtrans
 3180LOCALn%,l%,i%,j%,nl%,t1%,wrd$,max%:wrd$="":max%=0
 3190FORi%=0TO255:lu%?i%=0:NEXT
 3200FORi%=0TOMAXLOC%-1:FORj%=0TONTRANS%-1
 3210loctab%?(10*i%+j%)=&FF:NEXTj%:NEXTi%
 3220REPEAT:l%=FNritem(LOC%ORNOITEM%):IFl%=CEOF%UNTILTRUE:
GOTO3350
 3230IFl%<0THEN3330
```

```
 3240IF1%>max%max%=1%
 3250PROCslu(1%,1):n1%=0:REPEAT:n%=FNritem(MVERB%ORNOITEM%
)
 3260IFn%<0THENUNTILTRUE:GOTO3320
 3270t1%=FNritem(LOC%)
 3280IFt1%<0n%=t1%:UNTILTRUE:GOTO3320
 3290PROCslu(t1%,2):loctab%?(10*1%+n1%)=n%:loctab%?(10*1%+
n1%+5)=t1%
 3300n1%=n1%+1:IFn1%>NTRANS%PROCterr(2,1%,wrd$):UNTILTRUE:
GOTO3320
 3310UNTILFALSE
 3320IF(n%>0)AND(n1%=0)PROCterr(3,1%,"")
 3330IFn%=CEOF%UNTILTRUE:GOTO3350
 3340PROCslin:UNTILFALSE
 3350PROCtc:FORi%=1TOmax%
 3360IFlu%?i%=1PROCterr(4,i%,"")
 3370IFlu%?i%=2PROCterr(5,i%,"")
 3380IFlu%?i%=0PROCoutc(0):GOTO3430
 33901%=-1:REPEAT:1%=1%+1:UNTIL(loctab%?(10*i%+1%)=&FF)OR1
%=5
 3400PROCoutc(1%):IF1%>01%=1%-1
 3410FORj%=0TO1%:PROCoutc(loctab%?(10*i%+j%)):NEXT
 3420FORj%=0TO1%:PROCoutc(loctab%?(10*i%+j%+5)):NEXT
 3430NEXT:PROCoutc(&FF):ENDPROC
 3440DEFPROCserr(w$):PRINTw$," in line",L%:ENDPROC
 3450DEFFNgitem(k%,flag%):LOCALn%:n%=FNritem(k%ORNOITEM%)
 3460IFn%=CEOL%PROCslin:=FNgitem(k%,flag%)
```

```
 3470IF(NOTflag%)AND(n%=CEOF%)PRINT"syntax: eof":END
 3480IFn%>=1000PROCout(FNidnum("?")):n%=n%-1000
 3490PROCout(n%):=n%
 3500DEFFNidnum(s$):LOCALn%:n%=FNlookup(s$,FALSE):=FNgr1(n
%)
 3510DEFPROCsitem(s$):LOCALn%,i%:n%=FNidnum(s$)
 3520REPEAT:i%=FNgitem(NNOTYP%,TRUE)
 3530IFi%=CEOF%PRINT"eof?":END
 3540IFi%=n%UNTILTRUE:ENDPROC
 3550UNTILFALSE
 3560DEFPROCout(c%):outbuf%?O%=c%:O%=O%+1:ENDPROC
 3570DEFPROCpverb(flag%):LOCALi%,v%:v%=?outbuf%
 3580IFvtab%(v%)=-1vtab%(v%)=F%
 3590O%=O%-1:IFflag%THENoutbuf%?2=O%
 3600FORi%=1TOO%:BPUT#vchan%,outbuf%?i%:NEXT
 3610F%=F%+O%:O%=-flag%:ENDPROC
 3620DEFPROCsyntax:LOCALn%,v%,k%:PROCtc
 3630REPEAT:v%=FNgitem(VERB%,TRUE)
 3640IFv%=CEOF%UNTILTRUE:GOTO3790
 3650IFv%<0PROCsitem("END"):UNTILFALSE
 3660REPEAT:IFv%<TLIM%THEN3730
 3670n%=FNgitem(OBJ%,FALSE)
 3680IFn%<0UNTILTRUE:GOTO3770
 3690PROCout(0):n%=FNgitem(RESW%,FALSE)
 3700IF(n%=FNidnum("INVENT"))OR(n%=FNidnum("LOCAL"))THEN37
30
 3710IFn%=FNidnum("SKIP")THEN3730
```

```
 3720PROCserr("precondition expected"):UNTILTRUE:GOTO3770
 3730n%=FNstat(FNgitem(RESW%,FALSE))
 3740IFn%=FNidnum("END")PROCpverb(FALSE):UNTILTRUE:GOTO377
0
 3750IF(n%<>FNidnum("NEXT"))OR(v%<TLIM%)PROCserr("statemen
t?"):UNTILTRUE:GOTO3770
 3760PROCpverb(TRUE):UNTILFALSE
 3770IFn%<>FNidnum("END")PROCsitem("END")
 3780UNTILFALSE
 3790FORi%=0TO250:k%=FNgh1(i%):IF(k%<64)OR(k%>=192)THEN381
0
 3800IFvtab%(k%)=-1PRINThstr$(i%)," has no syntax"
 3810NEXT:PROCtc:k%=0:n%=-1:REPEAT:n%=n%+1:IFn%>191PROCout
c(&FD):UNTILTRUE:ENDPROC
 3820IFvtab%(n%)=-1k%=k%+1:GOTO3860
 3830IFk%=1PROCoutc(&FF):GOTO3850
 3840IFk%>1PROCoutc(&FE):PROCoutc(k%)
 3850PROCoutc((vtab%(n%)DIV256)AND&FF):PROCoutc(vtab%(n%)M
OD256):k%=0
 3860UNTILFALSE:ENDPROC
 3870DEFFNstat(nid%):LOCALn%,t%:n%=nid%:t%=0
 3880REPEAT:IFn%<OUNTILTRUE:=CERR%
 3890IFn%=FNidnum("IF")n%=FNifstat:GOTO3940
 3900IFn%=FNidnum("CASE")n%=FNcstat:GOTO3940
 3910IF(n%>=ALIM%)AND(n%<=RLIM%)n%=FNaction(n%):GOTO3940
 3920IFt%=0UNTILTRUE:=CERR%
 3930UNTILTRUE:=n%
```

```
 3940IFn%<0UNTILTRUE:=n%
 3950n%=FNgitem(RESW%,FALSE):t%=t%+1:UNTILFALSE
 3960DEFFNifstat:LOCALn%:n%=FNcond:IFn%<>0PROCsitem("FI"):
=CERR%
 3970n%=FNstat(FNgitem(RESW%,FALSE))
 3980REPEAT:IFn%<>FNidnum("ELIF")UNTILTRUE:GOTO4010
 3990n%=FNcond:IFn%=0n%=FNstat(FNgitem(RESW%,FALSE))
 4000UNTILFALSE
 4010IFn%=FNidnum("ELSE")n%=FNstat(FNgitem(RESW%,FALSE))
 4020IFn%=FNidnum("FI")THEN=0
 4030=CERR%
 4040DEFFNcond:LOCALn%,par%,idd%,k%:REPEAT:n%=FNgitem(COND
%,FALSE)
 4050IFn%=FNidnum("NOT")n%=FNgitem(COND%,FALSE)
 4060IF(n%<CLIM%)OR(n%>ALIM%)UNTILTRUE:=CERR%
 4070n%=n%-CLIM%:par%=0
 4080REPEAT:k%=sact%?(51+par%+3*n%):IFk%=0UNTILTRUE:GOTO41
10
 4090idd%=FNgitem(k%,FALSE):IFidd%<0UNTILTRUE:UNTILTRUE:=C
ERR%
 4100par%=par%+1:UNTILFALSE
 4110n%=FNgitem(RESW%,FALSE):IFn%=FNidnum("THEN")UNTILTRUE
:=0
 4120IF(n%<>FNidnum("AND"))AND(n%<>FNidnum("OR"))UNTILTRUE
:=CERR%
 4130UNTILFALSE
 4140DEFFNcstat:LOCALn%:n%=FNgitem(VAR%,FALSE):IFn%<0THEN4
```

```
200
 4150n%=FNgitem(RESW%,FALSE):REPEAT:IFn%=FNidnum("ESAC")UN
TILTRUE:=0
 4160IF(n%<>FNidnum("OF"))AND(n%<>FNidnum("DEFAULT"))UNTIL
TRUE:GOTO4200
 4170IFn%=FNidnum("OF")n%=FNcsel
 4180IFn%<0UNTILTRUE:GOTO4200
 4190n%=FNstat(FNgitem(RESW%,FALSE)):UNTILFALSE
 4200PROCsitem("ESAC"):=CERR%
 4210DEFFNcsel:LOCALn%:REPEAT:n%=FNgitem(VVAL%,FALSE):IFn%
<0THEN4230
 4220n%=FNgitem(RESW%,FALSE):IFn%=FNidnum(":")UNTILTRUE:=0
 4230IFn%<>FNidnum(",")UNTILTRUE:=CERR%
 4240UNTILFALSE
 4250DEFFNaction(nid%):LOCALn%,par%,idd%,k%:n%=nid%:IFn%<0
THEN=CERR%
 4260n%=n%-ALIM%:par%=0:REPEAT:k%=sact%?(3*n%+par%):IFk%=0
UNTILTRUE:=0
 4270idd%=FNgitem(k%,FALSE):IFidd%<0UNTILTRUE:=CERR%
 4280par%=par%+1:UNTILFALSE
 4290DEFPROCcl:FORI%=&482TO&4F4STEP2:IFI%=&4B6I%=I%+8
 4300A%=I%:REPEAT:B%=A%:A%=(!A%)AND&FFFF:UNTILA%<LOMEM ORA
%>V%
 4310?B%=0:B%?1=0:NEXT:!&4F6=0:?2=V%MOD256:?3=V%DIV256:END
PROC
 4320DEFPROCtc:LOCALp%:p%=PTR#vchan%+&1300
 4330BPUT#tchan%,p%MOD256:BPUT#tchan%,p%DIV256:ENDPROC
```

```
 4340DEFPROCoutc(c%):BPUT#vchan%,c%:ENDPROC
 4350DEFPROCtc2(a%):BPUT#tchan%,a%MOD256:BPUT#tchan%,a%DIV
256:ENDPROC
```

APPENDIX G

The interpreter

The interpreter resides in the following areas of memory:
&400 to &7FF
&840 to &CFF
&F00 to &12FF
and it uses zero page addresses 0 to &8F as workspace.

The various bits and pieces are loaded into memory by the loader.
The function of the loader is to:
–disable the Beeb's sound system
–load the interpreter into memory
–prompt for the directory in which the adventure can be found

It then passes control to a second loader, which will load the adventure itself, shift it down in memory and pass control to the interpreter.

The code of the interpreter starts with the definition of the pointers to the various tables generated by the scanner. They are:

base1	the huffman tree
base2	string pointer table section 0
base3	string pointer table section 1
base4	string pointer table section 2
base5	user hash table, strings
base6	hash table, id field
base7	transition table
base8	object table (the last section of the dictionary)
base9	verbs pointer table
base10	the encoded syntax file
base11	the encoded string file

The following ones point to internal workspace areas:

base12	the variable table (SCORE, CURLOC etc.)
base13	input buffer
base14	the visits array (keeps track of which locations the
player	has visited)
base15	the alarms table (max 10)
base16	the table of interpreter messages

Some of these tables have been compressed in order to save space.

They are:

- the hash table, strings field. An entry consists of:
  - a length byte (max 6) followed by the string
  - a zero byte, indicating one empty slot in the table
  - an &FF byte, followed by a count byte, indicating a series of empty slots
  - an &FE byte, indicating the end of the table

- the hash table, id field. All valid entries are simply stored consecutively, with all empty slots deleted

- the verbs pointer table. An entry consists of:
  - an offset word (2 bytes) with the high byte first
  - an &FF byte, indicating one empty slot
  - an &FE byte, followed by a count byte, indicating a series of empty slots
  - an &FD byte, indicating the end of the table

- the transition table. An entry consists of:
  - a count byte, indicating the number of transitions possible (max 5). This is followed by count bytes verb id's, and by count byte destinations
  - a 0 byte indicating an undefined entry
  - an &FF byte, indicating the end of the table

We will now describe the interpreter code. As usual, A, X and Y stand for the corresponding 6502 registers.

First we have a number of routines that access various tables.

routine objtab
: on entry, Y = table entry number, X = subfield. On return, A = value of subfield.
  X is encoded as follows:
  0 initial location of object
  1 its class
  2 its article
  3 its string number
  4 its hash table entry number

routine ptrvcode
: on entry, Y = verb number. On exit, vcodeptr points to start of syntax for the verb, i.e. (vcodeptr) = (base10) + vtab(Y)

routines conditions and actions
: perform a jump to one of the predefined functions (chloc, chclass etc.). On entry, A = routine number (starting at 0)

routine message
prints one of the predefined messages. On entry, X = message number

routine usererror
prints a message in red, and prints an extra word if the interpreter couldn't understand this word

routine ptrloctab
on entry, Y = location number. On exit, loctabptr = address of Yth entry of transition table.

routine vartab
on entry, A = variable number (first section of dictionary). On exit, A = value of variable

routine svartab
On entry, Y = variable number, A = value. The routine sets vartab(Y) to A.

routine readin
reads a line from the keyboard into the input buffer. It also takes care of DELs. The maximum line length is 85 characters, which allows a series of commands to be entered on one line

routine lookup
if X = 0, this routine searches for a string in the hash table. On entry, Y = length of string. The string is stored from cwa+5 onwards. If X <> 0, it jumps to lookupB, and Y = entry number in the hash table. On exit, stpt1 points to the string in hashtab(Y)

routine copystr
copies a string to another one. On entry, X = 0 means that the string is zero terminated, otherwise X = the length of the string

routine strcmp
compares two strings. On exit, the carry flag is set if the strins are equal

We now come to the routines that execute the adventure.

routine scancommand
this is the basic routine of the interpreter. It reads a line from the keyboard, looks up each word in the hash table, and decides wheter or not it makes sense what the player has typed.

The location sccom+1 may have the following values:
0 start of command
1 transitive verb encountered
2 tansitive verb + object encountered

routine trans
: this routine is called whenever a verb is typed which may cause a transition (north, enter etc.). It will perform the transition (if it is unconditional), print the description of the new location, update the score if necessary and check whether any alarms have gone off.

routine firstvisit
: checks whether or not it is the first time that the player visits the location. If so, it increments the score by 1

routine turns
: decrements all non-zero alarms by 1 and executes the alarms that reach zero

routine listloc
: prints a location description string

These routines are followed by the ones that implement the various keywords. These routines have, for obvious reasons, the same name as the corresponding keywords. They start with 'present' and end with 'chlstr'.

routine printstr
: this routine decodes the Huffman Code and prints the corresponding ASCII characters on the screen

routine proces
: executes a (verb, object) pair. In the case of an intransitive verb, object = 0. It locates the4 code for the verb in the syntax section and executes this code. It uses the routines 'statement', 'ifstat', 'skipthen', 'skipfi', 'cond', 'casestat', 'skipcase' and 'skipesac' to do this. Each of these routines is responsible for its own section of the syntax code

Finally we come to the main entry point of the interpreter, in which some of the tables are initialised and the description of location 1 is printed. This is followed by the error messages printed by the interpreter.

The very last bit of the interpreter code is dedicated to the second loader, which loads the two user adventure files, shifts them down in memory, and also contains a jump table for the various keywords.

```
    1 REM! Adventure Interpreter
    2 REM! ----------------------------
    3 REM! A.F. de Geus, J.H. Jongejan
    4 REM! and A.M. Koelmans 1985
    5 REM! ----------------------------
    6 REM! Program subject to copyright
    7 REM! ----------------------------
   10base1=&14:base2=&E:base3=&10:base4=&12:base5=&2:base6
=&4:base7=&6:base8=&0
   20base9=&A:base10=&8:base11=&C:base12=&16:base13=&18:ba
se14=&1A:base15=&1C:base16=&1E
   30stpt1=&20:stpt2=&22:scr1=&60:scr2=&62:scr3=&64:scr4=&
66
   40lookvar=&24:vcodeptr=&26:loctabptr=&28:listvar=&2A:rw
a=&2C
   50variable=&31:objlocvar=&33:lobjid=&34:lverbid=&35:alv
erb=&36:altimes=&37
   60lverb=&38:lobject=&3A:iwa=&3C:reprocvar=&40:ranvar=&4
2:slpvar=&44
   70cwa=&46:procvar=&51:visit=&57:transvar=&5A:sccom=&5D:
table=&68
   80OSWRCH=&FFEE:OSRDCH=&FFE0:OSNEWL=&FFE7:OSASCII=&FFE3:
OSKLI=&FFF7:OSBYTE=&FFF4
   90 OBJLIM=192:TLIM=128:ALLMOV=0:CLAY=254
  100CURLOC=0:OLDLOC=1:SCORE=2:CUROBJ=3:NTURNS=4:DUMMY=5:N
TRANS=5
  110IIF=201:TTHEN=202:ELIF=203:EELSE=204:FI=205:CASE=206:
```

```
OF=207
  120DDEFAULT=208:ESAC=209:NNEXT=210:EEND=211:NNOT=212:AAN
D=213:OOR=214
  130COMMA=216:DEREF=217:COLON=218:INVENT=230:LLOCAL=231:S
KIP=232
  140@%=6
  150CODE2=&840
  160DIM CODE1 &3FF,CODE3 &3FF,CODE5 150
  170FOR I%=4 TO6 STEP 2:P%=&400:O%=CODE1
  180[OPTI%
  190.objtab LDA base8:STA scr1:LDA base8+1:STA scr1+1:TYA
  200CMP #14:BCC g011:SEC:SBC #14:TAY:LDA #70:CLC:ADC scr1
:STA scr1
  210LDA scr1+1:ADC #0:STA scr1+1:TYA
  220.g011 STY scr2:ASL A:ASL A:CLC:ADC scr2:STX scr2:ADC
scr2
  230TAY:LDA (scr1),Y:RTS
  240
  250.ptrvcode LDA base9:STA scr1:LDA base9+1:STA scr1+1:L
DX #0:STY scr2
  260.g021 LDY #0:CPX scr2:BNE g022:LDA(scr1),Y:STA scr4+1
:INY
  270LDA (scr1),Y:STA scr4:LDA base10:CLC:ADC scr4:STA vco
deptr
  280LDA base10+1:ADC scr4+1:STA vcodeptr+1:RTS
  290.g022 BCC g023:LDA #&FF:RTS
  300.g023 LDA (scr1),Y:CMP #&FF:BNE g024:INX:LDA #1:JSR 1
```

```
nscr1:JMP g021
  310.g024 CMP #&FE:BNE g025:INY:LDA (scr1),Y:JSR incrx:LD
A #2:JSR inscr1:JMP g021
  320.g025 CMP #&FD:BNE g026:LDA #&FF:RTS
  330.g026 LDA #2:JSR inscr1:LDA #1:JSR incrx:JMP g021
  340
  350.conditions CLC:ADC #17
  360.actions ASL A:TAX:LDA table,X:STA scr1:INX:LDA table
,X:STA scr1+1:JMP (scr1)
  370
  380.message DEX:TXA:ASL A:TAX:LDA mesbase,X:STA stpt1
  390INX:LDA mesbase,X:STA stpt1+1:LDX #0
  400.mess LDY #0:CPX #0:BNE g044
  410.g041:LDA (stpt1),Y:JSR OSWRCH:BEQ g042:INY:JMP g041:
.g042:RTS
  420.g044 LDA (stpt1),Y:JSR OSWRCH:DEX:BEQ g042:INY:JMP g
044
  430.usererror JSR red:JSR message:LDA stpt2:STA stpt1
  440LDA stpt2+1:STA stpt1+1:JSR mess:JSR OSNEWL:RTS
  450.red LDA #129:JSR OSWRCH:RTS
  460
  470.ptrloctab LDA base7:STA scr1:LDA base7+1:STA scr1+1:
LDX #1:STY scr2
  480.g051 LDY #0:CPX scr2:BNE g052:LDA (scr1),Y:TAY:LDA #
1:JSR inscr1
  490LDA scr1:STA loctabptr:LDA scr1+1:STA loctabptr+1:TYA
:RTS
```

```
  500.g052 BCC g053:LDA #&FF:RTS
  510.g053 LDA (scr1),Y:CMP #&FF:BNE g054:RTS
  520.g054 CMP #0:BNE g055
  530.g056 INX:LDA #1:JSR inscr1:JMP g051
  540.g055 LDA (scr1),Y:ASL A:JSR inscr1:JMP g056
  550
  560.vartab TAY:LDA (base12),Y:RTS
  570.svartab STA (base12),Y:RTS
  580
  590.readin LDY #&FF:.g071:INY:CPY #84:BNE g072
  600 .g073 JSR OSRDCH:CMP #13:BNE g073:JSR OSASCII:.g076:
LDA #0:STA (base13),Y:RTS
  610.g072 JSR OSRDCH:CMP #13:BNE g074:JSR OSASCII:JMP g07
6
  620.g074 CMP #127:BNE g075:DEY:CPY #&FF:BEQ g071:DEY:JSR
 OSASCII:JMP g071:.g075:STA (base13),Y:JSR OSASCII:JMP g07
1
  630.lookup LDA #0:STA lookvar:LDA base5:STA scr1:LDA bas
e5+1:STA scr1+1
  640STY scr2:CPX #0:BEQ pr011:JMP lookupB
  650.pr011 LDY #0:LDA (scr1),Y:BNE pr012:LDA #1:JSR inscr
1:JMP pr011
  660.pr012 CMP #&FE:BNE pr013:LDA #&FF:RTS
  670.pr013 CMP #&FF:BNE pr014:LDA #2:JSR inscr1:JMP pr011
  680.pr014 CMP scr2:BNE pr015:PHA:LDA #1:JSR inscr1:LDA #
cwa+4:STA stpt1:LDA #0:STA stpt1+1:LDA scr1:STA stpt2
  690LDA scr1+1:STA stpt2+1:LDX cwa+3:JSR strcmp:PLA:BCC p
```

```
r018:LDY lookvar:LDA (base6),Y:RTS
  700.pr015 TAY:INY:TYA:.pr018:JSR inscr1:INC lookvar:JMP
pr011
  710
  720.lookupB LDX #0
  730.pr01B1 LDY #0:CPX scr2:BNE pr01B2:LDY #0:LDA (scr1),
Y:TAX
  740LDA scr1:STA stpt1:LDA scr1+1:STA stpt1+1:INC stpt1:B
NE pr01B6:INC stpt1+1:.pr01B6:RTS
  750.pr01B2 BCC pr01B3:LDA #&FF:RTS
  760.pr01B3 LDA (scr1),Y:BNE pr01B4
  770.pr01B7 INX:LDA #1:JSR inscr1:JMP pr01B1
  780.pr01B4 CMP #&FE:BNE pr01B5:LDA #&FF:RTS
  790.pr01B5 CMP #&FF:BNE pr01B8:INY:LDA (scr1),Y:JSR incr
x:LDA #2:JSR inscr1:JMP pr01B1
  800.pr01B8 TAY:STY lookvar+1:INY:TYA:JSR inscr1:INC look
var:INX:JMP pr01B1
  810
  820.inscr1 CLC:ADC scr1:STA scr1:BCC pr017:INC scr1+1:.p
r017:RTS
  830.incrx STX scr4:CLC:ADC scr4:TAX:RTS
  840
  850.copystr LDY #0:CPX #0:BNE pr024C
  860.pr021C LDA (stpt1),Y:STA (stpt2),Y:BEQ pr022C:INY:JM
P pr021C
  870.pr022C INY:LDA #0:STA(stpt2),Y:RTS
  880.pr024C LDA (stpt1),Y:STA (stpt2),Y:DEX:BEQ pr022C:IN
```

```
Y:JMP pr024C
  890
  900.strcmp LDY #0:CPX #0:BNE pr034C
  910.pr031C LDA (stpt1),Y:CMP (stpt2),Y:BNE pr032C:INY:TA
X:BNE pr031C:SEC:RTS
  920.pr032C CLC:RTS
  930.pr034C LDA (stpt1),Y:CMP (stpt2),Y:BNE pr032C:INY:DE
X:BNE pr034C:SEC:RTS
  940.scancommand
  950LDA #0:STA sccom+1:STA sccom+2:LDA lverbid:CMP #TLIM:
BCC E10:LDA #2:STA sccom+1
  960.E10 JSR readin
  970.E11 LDY sccom+2:LDA (base13),Y:CMP #ASC(" ")+1:BCS E
15:INY:STY sccom+2:CMP #0:BNE E11
  980LDA sccom+1:CMP #1:BEQ E13:RTS
  990.E13 LDA #cwa+4:STA stpt2:LDA #0:STA stpt2+1:LDX #1:J
SR usererror:RTS
 1000.E15 LDX #0
 1010.E16 LDA (base13),Y:CMP #ASC(" ")+1:BCC E120:CPX #6:B
CS E17:STA cwa+4,X:INX
 1020.E17 INY:STY sccom+2:JMP E16
 1030.E120 LDA #0:STA cwa+4,X:STX cwa+3
 1040LDX #0:LDY cwa+3:JSR lookup:STA sccom
 1050LDA #cwa+4:STA stpt2:LDA #0:STA stpt2+1
 1060LDA sccom:CMP #&FF:BNE E130:LDA #0:STA sccom+1:LDX #2
:JSR usererror:JMP E11
 1070.E130 LDA sccom:CMP #192:BCC E160:LDA sccom+1:BNE E14
```

```
0:LDX #3:JSR usererror:JMP E11
 1080.E140 LDA sccom:STA lobjid:LDA #cwa+4:STA stpt1:LDA #
0:STA stpt1+1
 1090LDA lobject:STA stpt2:LDA lobject+1:STA stpt2+1:LDX #
0:JSR copystr
 1100LDA lverbid:PHA:LDA lobjid:PHA:JSR proces:LDA #2:STA
sccom+1:JMP E11
 1110.E160 STA lverbid:LDA #cwa+4:STA stpt1:LDA #0:STA stp
t1+1:LDA lverb:STA stpt2
 1120LDA lverb+1:STA stpt2+1:LDX #0:JSR copystr:LDA sccom:
CMP #TLIM:BCC E170:LDA #1:STA sccom+1:JMP E11
 1130.E170 LDA #0:STA sccom+1:LDA sccom:CMP #64:BCC E180:P
HA:LDA #0:PHA:JSR proces:JMP E11
 1140.E180 STA transvar:JSR trans:JMP E11
 1150
 1160.trans LDA #CURLOC:JSR vartab:STA transvar+1:TAY:JSR
ptrloctab:STA transvar+2
 1170LDY #OLDLOC:LDA transvar+1:JSR svartab:LDA #0:STA tra
nsvar+1
 1180.E20 LDA transvar+1:CMP transvar+2:BCS E200:TAY
 1190LDA (loctabptr),Y:CMP #ALLMOV:BNE E21:LDX #ALLMOV:STX
 transvar
 1200.E21 CMP transvar:BNE E250:TYA:CLC:ADC transvar+2:TAY
:LDA (loctabptr),Y:BNE E22
 1210LDA transvar:PHA:LDA #0:PHA:JSR proces:JMP E23
 1220.E22 LDY #CURLOC:JSR svartab
 1230JSR turns:.E23:LDA #CURLOC:JSR vartab:TAY:JSR listloc
```

```
:JSR firstvisit:BCC E24
 1240LDA #SCORE:JSR vartab:CLC:ADC #1:LDY #SCORE:JSR svart
ab
 1250.E24 RTS
 1260.E250 INC transvar+1:JMP E20
 1270.E200 LDX #4:LDA lverb:STA stpt2:LDA lverb+1:STA stpt
2+1:JSR usererror:RTS
 1280]
 1290IFI%=6PRINT"*SAVE ADV1 "+~CODE1+","+~O%+" 400 400"
 1300P%=&840:O%=CODE2
 1310[OPTI%
 1320.firstvisit
 1330LDA #CURLOC:JSR vartab:STA visit:LSR A:LSR A:LSR A:ST
A visit+1
 1340LDA visit:AND #7:BNE E32:LDA #1:BNE E33:.E32:TAX:LDA
£1
 1350.E30 ASL A:DEX:BNE E30:.E33:STA visit:LDY visit+1:LDA
 (base14),Y:STA visit+2
 1360AND visit:BEQ E31:CLC:RTS
 1370.E31 LDA visit+2:ORA visit:STA (base14),Y:SEC:RTS
 1380
 1390.turns LDA #NTURNS:JSR vartab:CLC:ADC #1:LDY #NTURNS:
JSR svartab:LDA #0:STA altimes
 1400.E50 LDY altimes:CPY #22:BCS E59:LDA (base15),Y:BEQ E
58:SEC:SBC #1:STA (base15),Y
 1410BNE E58:INY:LDA (base15),Y:PHA:LDA #0:PHA:JSR proces
 1420.E58 INC altimes:INC altimes:BNE E50
```

```
 1430.E59 RTS
 1440.listloc STY listvar:JSR OSNEWL:LDA #1:JSR printstr:L
DA #1:STA listvar+1
 1450.p011 LDY listvar+1:CPY #50:BNE p0110:.p016:RTS
 1460.p0110 LDX #0:JSR objtab:TAX:TYA:JSR inscr1:LDA scr1+
1:CMP base5+1:BNE p017:LDA scr1:CMP base5:BCS p016
 1470.p017 TXA:CMP listvar:BNE p015:CMP #0:BNE p012
 1480LDX #9:JSR message:LDY listvar+1:LDX #4:JSR objtab:TA
Y
 1490LDX #1:JSR lookup:JSR mess:JSR OSNEWL
 1500.p015 INC listvar+1:JMP p011
 1510.p012 LDY listvar+1:LDX #3:JSR objtab:TAY:LDA #2:JSR
printstr:JMP p015
 1520
 1530.getitem LDY #0:LDA (vcodeptr),Y:INC vcodeptr:BNE p02
1:INC vcodeptr+1
 1540.p021 RTS
 1550
 1560.getobj JSR getitem:STA objlocvar:CMP #CLAY:BNE p031:
LDA lobjid:STA objlocvar
 1570.p031 LDA #CUROBJ:JSR vartab:CLC:ADC objlocvar:SEC:SB
C #OBJLIM:RTS
 1580
 1590.getval JSR getitem:CMP #DEREF:BNE p041:JSR getitem:J
SR vartab
 1600.p041 RTS
 1610
```

```
1620.present JSR getobj:TAY:LDX #0:JSR objtab:STA iwa:LDA
£CURLOC
1630JSR vartab:CMP iwa:BNE p051:SEC:RTS
1640.p051 CLC:RTS
1650
1660 .holding JSR getobj:TAY:LDX #0:JSR objtab:CMP #0:BNE
p051:SEC:RTS
1670
1680.testcl JSR getobj:TAY:LDX #1:JSR objtab:STA iwa:JSR
getval
1690CMP iwa:BNE p051:SEC:RTS
1700
1710 .equal JSR getval:JSR vartab:STA iwa:JSR getval
1720CMP iwa:BNE p081:SEC:RTS
1730.p081 CLC:RTS
1740
1750.smaller JSR getval:JSR vartab:STA iwa:JSR getval:CMP
iwa:BCS p091
1760.p092 CLC:RTS
1770.p091 BEQ p092:SEC:RTS
1780
1790.smequal JSR getval:JSR vartab:STA iwa:JSR getval:CMP
iwa:BCS p100:CLC:RTS
1800.p100 SEC:RTS
1810
1820.readb JSR OSRDCH:JSR OSASCII:CMP #ASC("y"):BNE p110
1830.p111 SEC:RTS
```

```
 1840.p110 CMP #ASC("Y"):BEQ p111:CLC:RTS
 1850
 1860.objstr JSR getobj:STA iwa:JSR getval:STA iwa+1:LDY i
wa
 1870LDX #3:JSR objtab:CMP iwa+1:BNE p120:SEC:RTS
 1880.p120 CLC:RTS
 1890
 1900.lastobj JSR getobj:STA iwa:LDA lobjid:SEC:SBC #OBJLI
M
 1910CMP iwa:BNE p130:SEC:RTS
 1920.p130 CLC:RTS
 1930
 1940.objloc JSR getobj:TAY:LDX #0:JSR objtab:STA iwa:JSR
getval
 1950CMP iwa:BNE p140:SEC:RTS
 1960.p140 CLC:RTS
 1970
 1980.chloc JSR getobj:STA iwa:JSR getval:STA iwa+1:LDY iw
a:LDX #0:JSR objtab
 1990LDA iwa+1:STA (scr1),Y:LDA #CURLOC:JSR vartab:CMP iwa
+1:BEQ p150:RTS
 2000.p150 LDY iwa:LDX #3:JSR objtab:TAY:LDA #2:JSR prints
tr:RTS
 2010
 2020.chclass JSR getobj:STA iwa:JSR getval:STA iwa+1:LDY
iwa
 2030LDX #1:JSR objtab:LDA iwa+1:STA (scr1),Y:RTS
```

```
 2040
 2050.chostr JSR getobj:STA iwa:JSR getval:STA iwa+1:LDY i
wa:LDX #3
 2060JSR objtab:LDA iwa+1:STA (scr1),Y:LDA #CURLOC:JSR var
tab:STA iwa+2
 2070LDY iwa:LDX #0:JSR objtab:CMP iwa+2:BEQ p170:RTS
 2080.p170 LDY iwa+1:LDA #2:JSR printstr:RTS
 2090
 2100.setran LDA &FE44:EOR &FE54:EOR &FE64:EOR &FE74:EOR &
FE78:STA rwa
 2130JSR getval:STA ranvar:JSR getval:STA ranvar+1:BEQ p18
09:LDA rwa
 2140.p181 CMP ranvar+1:BCC p1809:SBC ranvar+1:JMP p181
 2150.p1809 LDY ranvar:JSR svartab:RTS
 2160
 2170 .setvar JSR getval:STA iwa:JSR getval:LDY iwa:JSR sv
artab:RTS
 2180
 2190.add JSR getval:STA iwa:JSR vartab:STA iwa+1:JSR getv
al:CLC
 2200ADC iwa+1:LDY iwa:JSR svartab:RTS
 2210
 2220.sub JSR getval:STA iwa:JSR vartab:STA iwa+1:JSR getv
al:SEC
 2230SBC iwa+1:EOR #&FF:LDY iwa:JSR svartab:RTS
 2240
 2250.printi JSR getval:JSR vartab:STA cwa:LDA #0:STA cwa+
```

```
1:LDA #6:STA cwa+4
 2260.p220 LDA #0:STAcwa+2:STA cwa+3:LDX #16:CLC
 2270.p221 ROL cwa:ROL cwa+1:ROL cwa+2:ROL cwa+3:SEC:LDA c
wa+2:SBC #10
 2280TAY:LDA cwa+3:SBC #0:BCC p222:STY cwa+2:STA cwa+3
 2290.p222 DEX:BNE p221:ROL cwa:ROL cwa+1:LDA cwa+2:CLC:AD
C #ASC("0"):JSR outchar
 2300LDA cwa:ORA cwa+1:BNE p220:LDX #9:JSR message:LDX cwa
+4:INX
 2310.p223 LDA cwa+4,X:JSR OSWRCH:INX:CPX #7:BNE p223:JSR
OSNEWL:RTS
 2320.outchar LDX cwa+4:BNE p224:RTS
 2330.p224 STA cwa+4,X:DEC cwa+4:RTS
 2340
 2350.read1 JSR getval:STA cwa+3
 2360.p236 LDA #0:STA cwa:STA cwa+1:LDX #6
 2370 .p230 JSR OSRDCH:CMP #127:BNE p237:CPX #6:BEQ p230:I
NX:JSR OSASCII:BNE p230:.p237:JSR OSASCII
 2380CMP #13:BEQ p231:STA cwa+4,X:DEX:CPX #2:BNE p230
 2390JSR OSNEWL:JMP erfound1
 2400.p231 STX cwa+4:LDX #6
 2410.p233 LDA cwa+4,X:CMP #ASC("0"):BMI erfound2:CMP #ASC
("9")+1:BPL erfound2
 2420STA cwa+2:ASL cwa:ROL cwa+1:LDA cwa:LDY cwa+1:ASL cwa
:ROL cwa+1
 2430ASL cwa:ROL cwa+1:CLC:ADC cwa:STA cwa:TYA:ADC cwa+1:S
TA cwa+1
```

```
 2440 LDA cwa+2:SEC:SBC #ASC("0"):CLC:ADC cwa:STA cwa:BCC
p234:INC cwa+1
 2450.p234 DEX:CPX cwa+4:BNE p233:LDA cwa+1:BNE erfound1
 2460LDA cwa:LDY cwa+3:JSR svartab:RTS
 2470.erfound1 JSR red:LDX #10
 2480.p235 JSR message:JSR OSNEWL:JMP p236
 2490.erfound2 LDX #11:BNE p235
 2500
 2510.reproc JSR getval:STA procvar:JSR getobj:STA procvar
+1
 2520LDA vcodeptr:PHA:LDA vcodeptr+1:PHA:LDA procvar:PHA:L
DA procvar+1:PHA
 2530JSR proces:PLA:STA vcodeptr+1:PLA:STA vcodeptr:RTS
 2540
 2550.list1 JSR getval:TAY:JSR listloc:RTS
 2560
 2570.slp JSR getval:BNE p260:RTS
 2580.p260 STA slpvar
 2590.p262 LDX #10:SEI
 2600.p261 JSR delay:DEX:BNE p261:DEC slpvar:BNE p262:CLI:
RTS
 2610.delay PHA:TXA:PHA:LDX #&64
 2620.p263 LDA #&F8:STA slpvar+1
 2630.p264 DEC slpvar+1:BNE p264:DEC main:DEX:BNE p263:INC
 main:PLA:TAX:PLA:RTS
 2640
 2650.printa JSR getval:STA iwa:JSR getval:TAX:LDA iwa
```

```
 2660.p270 JSR OSWRCH:DEX:BNE p270:RTS
 2670
 2680.Message JSR getval:TAY:LDA #0:JSR printstr:RTS
 2690.quit LDX #f114 MOD 256:LDY #f114 DIV 256:JMP OSKLI:.
f114 EQUS "BASIC":EQUB 13
 2700
 2710.alarm LDY #0:STY alverb
 2720.p290 LDY alverb:CPY #22:BCS p299:LDA (base15),Y:BNE
p298:JSR getval:LDY alverb
 2730INY:STA (base15),Y:JSR getval:LDY alverb:STA (base15)
,Y:RTS
 2740.p298 INC alverb:INC alverb:BNE p290
 2750.p299 RTS
 2760
 2770.chlstr JSR getval:STA iwa:JSR getval:STA iwa+1:LDA b
ase3:STA scr1
 2780LDA base3+1:STA scr1+1:LDA iwa+1:JSR seta:LDA (scr1),
Y:STA iwa+2
 2790INY:LDA (scr1),Y:STA iwa+3:LDA base3+1:STA scr1+1:LDA
 iwa:JSR seta
 2800LDA iwa+2:STA (scr1),Y:INY:LDA iwa+3:STA (scr1),Y:RTS
 2810.seta CMP #128:BCS p301:ASL A:TAY:RTS
 2820.p301 AND #127:ASL A:TAY:INC scr1+1:RTS
 2830
 2840.printstr
 2850ASL A:TAX:LDA base2,X:STA scr1:INX:LDA base2,X:STA sc
r1+1:TYA:JSR seta
```

```
 2860LDA (scr1),Y:STA scr2:INY:LDA (scr1),Y:STA scr2+1:LDA
 base11:CLC:ADC scr2
 2870 STA scr2:LDA base11+1:ADC scr2+1:STA scr2+1:JSR gmem
:LDX #8:JMP newchar
 2880]
 2890IFI%=6PRINT"*SAVE ADV2 ",~CODE2,~O%," 840 840"
 2900P%=&F00:O%=CODE3
 2910[OPTI%
 2920.newchar LDA base1:STA scr1:LDA base1+1:STA scr1+1:LD
Y #0
 2930.nxtbit LDA (scr1),Y:INY:JSR gbit:BCC incadd:INY:INY
 2940.incadd JSR itree:LDY #0:LDA (scr1),Y:BNE outc:DEX:BN
E nxtbit:JSR gmem:JMP nxtbit
 2950.outc CMP #10:BEQ exit:JSR OSASCII:DEX:BNE tloop:JSR
gmem
 2960.tloop JMP newchar
 2970.exit JSR OSNEWL:RTS
 2980.gmem PHA:TYA:PHA:LDY #0:LDA (scr2),Y:STA scr3:INC sc
r2:BNE restor:INC scr2+1
 2990.restor PLA:TAY:PLA:LDX #8:RTS
 3000.itree PHA:LDA (scr1),Y:PHA:INY:LDA (scr1),Y:STA scr1
+1:PLA:STA scr1:PLA:RTS
 3010.gbit PHA:LDA scr3:ROL A:STA scr3:PLA:RTS
 3020
 3030.proces PLA:TAX:PLA:TAY:PLA:STA procvar:PLA:STA procv
ar+1:TYA:PHA:TXA:PHA
 3040LDY procvar+1:JSR ptrvcode:LDA procvar:SEC:SBC #OBJLI
```

```
M:STA procvar
 3050LDY #CUROBJ:LDA #0:JSR svartab:LDA procvar+1:CMP #TLI
M:BCS pr1030:JMP pr1100
 3060.pr1030 JSR getobj:STA procvar+2
 3070.pr1040 LDA procvar:CMP procvar+2:BEQ pr1070:JSR geti
tem:STA procvar+3:CMP £0:BNE pr1050
 3075LDX #5:JSR red:JSR message:RTS
 3080.pr1050 LDA procvar+3:SEC:SBC #2:CLC:ADC vcodeptr:STA
 vcodeptr:BCC pr1051:INC vcodeptr+1
 3090.pr1051 JSR getobj:STA procvar+2:JMP pr1040
 3100.pr1070 INC vcodeptr:BNE pr1071:INC vcodeptr+1
 3110.pr1071 JSR getitem:STA procvar+4:CMP #INVENT:BNE pr1
072
 3120LDY procvar+2:LDX #0:JSR objtab:BNE pr1080
 3130.pr1072 LDA procvar+4:CMP £LLOCAL:BNE pr1100:LDA #CUR
LOC:JSR vartab
 3140STA procvar+5:LDY procvar+2:LDX #0:JSR objtab:CMP pro
cvar+5:BEQ pr1100
 3150.pr1080 JSR red:LDA procvar+4:CMP #INVENT:BNE pr1081:
LDX #6:JSR message:JMP pr1082
 3160.pr1081 LDX #7:JSR message
 3170.pr1082 LDY procvar+2:LDX #2:JSR objtab:BEQ pr1084
 3180CMP #1:BNE pr1083:LDX #12:JSR message:JMP pr1084
 3190.pr1083 LDX #13:JSR message
 3200.pr1084 LDX #14:JSR message:LDA procvar+4:CMP £LLOCAL
:BNE pr1085:LDX #8:JSR message
 3210.pr1085 JSR OSNEWL:RTS
```

```
 3220.pr1100 JSR statement:JSR turns:RTS
 3230
 3240.statement
 3250 JSR getitem:CMP #IIF:BNE pr021:JSR ifstat:JMP statem
ent
 3260 .pr021 CMP #CASE:BNE pr022:JSR casestat:JMP statemen
t
 3270.pr022 CMP #151:BCC pr023:CMP #201:BCS pr023:SEC:SBC
£151:JSR actions:JMP statement
 3280 .pr023 DEC vcodeptr:BNE pr024:DEC vcodeptr+1
 3290 .pr024 RTS
 3300
 3310.ifstat JSR cond
 3320.pr031 BNE pr0310:JSR skipthen:BNE pr032:RTS
 3330.pr032 CMP #1:BNE pr033:LDA #1:JMP pr0310
 3340.pr033 JSR cond:JMP pr031
 3350.pr0310 BEQ pr034:JSR statement
 3360.pr034 JSR skipfi:RTS
 3370
 3380.skipthen
 3390JSR getitem:CMP #IIF:BNE pr041:JSR skipfi:JMP skipthe
n
 3400.pr041 CMP #FI:BNE pr042:LDA #0:RTS
 3410.pr042 CMP #EELSE:BNE pr043:LDA #1:RTS
 3420.pr043 CMP #ELIF:BNE skipthen:LDA #2:RTS
 3430
 3440.skipfi
```

```
 3450JSR getitem:CMP #FI:BNE pr051:RTS
 3460.pr051 CMP #IIF:BNE skipf1:JSR skipf1:JMP skipf1
 3470
 3480.cond LDA #0:STA scr3:JSR getitem:CMP #NNOT:BNE pr061
:INC scr3:JSR getitem
 3490.pr061 SEC:SBC #101:TAX:LDA scr3:PHA:TXA:JSR conditio
ns:BCC pr066:LDA #1:BNE pr065:.pr066 LDA #0:.pr065 STA scr
2:PLA:BEQ pr062
 3500LDA scr2:EOR #1:STA scr2
 3510.pr062 JSR getitem:CMP #AAND:BNE pr063:LDA scr2:PHA:J
SR cond:STA scr2:PLA:AND scr2:RTS
 3520.pr063 CMP #OOR:BNE pr064:LDA scr2:PHA:JSR cond:STA s
cr2:PLA:ORA scr2:RTS
 3530.pr064 LDA scr2:RTS
 3540
 3550.casestat
 3560JSR getitem:JSR vartab:STA scr1:JSR getitem
 3570.pr071 CMP #ESAC:BNE pr072:RTS
 3580.pr072 CMP #DDEFAULT:BNE pr073:JSR statement:JMP pr07
20
 3590.pr073 JSR getitem:CMP scr1:BNE pr0710
 3600.pr074 JSR getitem:CMP #COLON:BNE pr074:JSR statement
:JSR skipesac:RTS
 3610.pr0710 JSR getitem:CMP #COMMA:BEQ pr073:JSR skipcase
:JMP pr071
 3620.pr0720 JSR getitem:CMP #ESAC:BNE pr0720:RTS
 3630
```

```
 3640.skipcase
 3650JSR getitem:CMP #CASE:BNE pr081:JSR skipesac:JMP skip
case
 3660.pr081 CMP #OF:BEQ pr082:CMP #DDEFAULT:BEQ pr082:CMP
£ESAC:BEQ pr082:BNE skipcase
 3670.pr082:RTS
 3680
 3690.skipesac
 3700JSR getitem:CMP #CASE:BNE pr091:JSR skipesac:JMP skip
esac
 3710.pr091 CMP #ESAC:BNE skipesac:RTS
 3720.main LDA #200:LDX #1:LDY #0:JSR &FFF4
 3725JSR OSNEWL:LDY #50:LDA £0
 3730.m1 JSR svartab:DEY:BNE m1:JSR svartab:LDY #32
 3740.m2 STA (base14),Y:DEY:BNE m2:STA (base14),Y:LDY #20:
LDA #0
 3750.m4 STA (base15),Y:DEY:BNE m4:STA (base15),Y:LDA #1:L
DY #OLDLOC:JSR svartab
 3760LDY #CURLOC:JSR svartab:LDA #0:STA lverbid:STA lobjid
:LDY #1:JSR listloc
 3770LDA #&D:STA lverb+1:STA lobject+1:LDA #&60:STA lverb:
LDA #&A0:STA lobject
 3780.m3 JSR scancommand:JMP m3
 3790.mesbase
 3800EQUW mes1
 3810EQUW mes2
 3820EQUW mes3
```

```
3830EQUW mes4
3840EQUW mes5
3850EQUW mes6
3860EQUW mes7
3870EQUW mes8
3880EQUW mes9
3890EQUW mes10
3900EQUW mes11
3910EQUW mes12
3920EQUW mes13
3930EQUW &DA0
3940.mes1
3950EQUS "What or who must I "
3960EQUB 0
3970.mes2
3980EQUS "I don't know the word"
3990EQUB 131:EQUB 0
4000.mes3
4010EQUS "What must I do with "
4020EQUB 0
4030.mes4
4040EQUS "There is no way "
4050EQUB 0
4060.mes5
4070EQUS "impossible"
4080EQUB 0
4090.mes6
```

```
4100EQUS "You are not carrying "
4110EQUB 0
4120.mes7
4130EQUS "I don't see "
4140EQUB 0
4150.mes8
4160EQUS " here"
4170EQUB 0
4180.mes9
4190EQUS "        "
4200EQUB 0
4210.mes10
4220EQUS "A value between 0 and 255 please"
4230EQUB 0
4240.mes11
4250EQUS "A number please"
4260EQUB 0
4270.mes12
4280EQUS "an "
4290EQUB 0
4300.mes13
4310EQUS "a "
4320EQUB 0
4330]
4340IFI%=6PRINT"*SAVE ADV3 ",~CODE3,~O%," 1300 1300"
4350P%=&12:O%=CODE5
4360[OPTI%
```

```
 4370.fil2 EQUS"LOAD ?.1 1700":EQUB 13
 4380.fil3 EQUS"LOAD ?.2 0":EQUB 13
 4390.ld2 LDX #fil2 MOD 256:LDY #0:JSR OSKLI:LDX #fil3 MOD
256:LDY #0:JSR OSKLI
 4400LDY #0:STY ld2:STY ld2+2:LDA #&F:STA ld2+1:LDX #&13:S
TX ld2+3
 4410.lloop LDA (ld2+2),Y:STA (ld2),Y:INY:BNE lloop
 4420INC ld2+1:INC ld2+3:INX:CPX #&80:BNE lloop
 4430LDA #12:JSR OSWRCH:LDA £mesbase MOD256:STA base16:LDA
 £mesbase DIV256:STA base16+1:JMP main:BRK
 4440EQUW chloc
 4450EQUW chclass
 4460EQUW chostr
 4470EQUW setran
 4480EQUW setvar
 4490EQUW add
 4500EQUW sub
 4510EQUW printi
 4520EQUW readi
 4530EQUW reproc
 4540EQUW listl
 4550EQUW Message
 4560EQUW slp
 4570EQUW printa
 4580EQUW quit
 4590EQUW chlstr
 4600EQUW alarm
```

```
4610EQUW present
4620EQUW holding
4630EQUW testcl
4640EQUW equal
4650EQUW smaller
4660EQUW smequal
4670EQUW readb
4680EQUW objstr
4690EQUW lastobj
4700EQUW objloc
4710]
4720IFI%=6PRINT"*SAVE ADV4 "+~CODE5+~O%+" 2B 12"
4730NEXT
4740END
```



```
    1 REM! loader for adventure system
    2 REM! ----------------------------
    3 REM! A.F. de Geus, J.H. Jongejan
    4 REM! and A.M. Koelmans 1984
    5 REM! ----------------------------
    6 REM! Program subject to copyright
    7 REM! To run, type "*adv"
    8 REM! ----------------------------
   10base1=&14:base2=&E:base3=&10:base4=&12:base5=&2:base6
=&4:base7=&6:base8=&0
```

```
   20base9=&A:base10=&8:base11=&C:base12=&16:base13=&18:ba
se14=&1A:base15=&1C:base16=&1E
   30stpt1=&20:stpt2=&22:scr1=&60:scr2=&62:scr3=&64:scr4=&
66
   40lookvar=&24:vcodeptr=&26:loctabptr=&28:listvar=&2A:rw
a=&2C:ld2=&2B:fil2=&12:fil3=&20
   50variable=&31:objlocvar=&33:lobjid=&34:lverbid=&35:alv
erb=&36:altimes=&37
   60lverb=&38:lobject=&3A:iwa=&3C:reprocvar=&40:ranvar=&4
2:slpvar=&44
   70cwa=&46:procvar=&51:visit=&57:transvar=&5A:sccom=&5D:
table=&68
   80OSWRCH=&FFEE:OSRDCH=&FFE0:OSNEWL=&FFE7:OSASCII=&FFE3:
OSKLI=&FFF7:OSBYTE=&FFF4
   90 OBJLIM=192:TLIM=128:ALLMOV=0:CLAY=254
  100CURLOC=0:OLDLOC=1:SCORE=2:CUROBJ=3:NTURNS=4:DUMMY=5:N
TRANS=5
  110IIF=201:TTHEN=202:ELIF=203:EELSE=204:FI=205:CASE=206:
OF=207
  120DDEFAULT=208:ESAC=209:NNEXT=210:EEND=211:NNOT=212:AAN
D=213:OOR=214
  130COMMA=216:DEREF=217:COLON=218:INVENT=230:LLOCAL=231:S
KIP=232
  140@%=6
  150DIM CODE4 130
  160FOR I%=4 TO6 STEP 2
  170P%=&1900:O%=CODE4
```

```
180[OPTI%
190.fil EQUS"LOAD ADV1":EQUB 13
200LDY #0:LDX #1:LDA #210:JSR OSBYTE
210LDX #fil MOD256:LDY #fil DIV256:JSR OSKLI
220INC fil+8:LDX #fil MOD256:LDY #fil DIV256:JSR OSKLI
230INC fil+8:LDX #fil MOD256:LDY #fil DIV256:JSR OSKLI
240INC fil+8:LDX #fil MOD256:LDY #fil DIV256:JSR OSKLI
250 LDA #12:JSR OSWRCH:JSR OSNEWL
260LDA #ASC("D"):JSR OSWRCH
270LDA #ASC("I"):JSR OSWRCH
280LDA #ASC("R"):JSR OSWRCH
290LDA #ASC(":"):JSR OSWRCH
300JSR &FFE0:BCC 11:LDY #0:LDX #0:LDA #210:JSR OSBYTE
310LDA #7:JSR OSWRCH:RTS
320.11 JSR OSWRCH:STA fil2+5:STA fil3+5:JSR OSNEWL
330JMP 1d2:BRK
340]
350IFI%=6PRINT"*SAVE ADV  ",~CODE4,~O%," 190A 1900"
360NEXT
370END
```

INDEX

Action Procedures 24
Acyclic Graphs 111
Adams, Scott 1
ADD 25
ALARM 25
Alarms 53
ALLMOV 19
Alphabet, input 90
Alphabet, output 90
AND 23
Arc 106
ASCII 10
ASCII Code 114
Automaton 90
Automaton, finite 91

Backus- Naur Formalism 87
BNF 87
Boolean 23

CASE 23
CHCLASS 25
CHLOC 25
CHLSTR 25
Chomsky, N 85
CHOSTR 25
CLAY 19
Colossal Adventure 73, 114
Compiler 2, 12
Completeness 109
Compression, text 114, 10
Condition Functions 24
Conditional Transitions 16
Conditions 23
Connectedness 111
Context Free Grammar 85
Context Sensitive Grammar 85
Crowther, W 1
CURLOC 18, 37
Cyclic Graphs 111

Deferencing 18
Dictionary 7, 29
Digraph 106
DUMMY 18

Edge 106
Editor, screen 156
EQUAL 26
Error Messages 153
Errors 104
Execeutable Image 12

Finite Automaton 91
Finite State Generator 93
Finite State Recogniser 96

FORTRAN 1

Generators 93
Grammar 79, 80, 103
Grammar, abstract definition of 82
Grammar, context free 85, 103
Grammar, context sensitive 85
Grammar, regular 85
Grammar, type 0 86
Graphs 106
Graphs, acyclic 111
Graphs, cyclic 111
Graphs, directed 106
Graphs, planar 113

Hamiltonian Paths 109
Hashing 7, 119
Hashing, closed 112
Hashing, linear 122
Hashing, open 121
Help Commands 56
HOLDING 26
Huffman Code 10, 114
Huffman Tree 117

IF 22
Interpreted Language 1
Interpreter 2, 12, 36, 105
Interpreter, listing of 203
Intransitive Verbs 5, 15
INVENT 19

LASTOBJ 26
LISTLOC 25
LOCAL 19

Map 41, 9, 41
MESSAGE 25
Messages 5
Metalanguage 87
Mission Impossible 1

Nodes 106
Nondeterministic Machines 97
NOT 23
NTURNS 18

Objects 5, 15, 17, 5, 16
OBJLOC 26
OBJSTR 26
OLDLOC 18, 37
OR 23
Output Alphabet 90

Parse Tree 84
Parsing 83
Path 106
Paths, Hamiltonian 109
Paths, length of 107
Pirate Adventure 1
Portability 11

PRESENT 26
PRINT 25
PRINTA 25

QUIT 25

Random Events 71
Random Transitions 16, 71
READ 25
READB 26
Recognisers 93
Recursion 76
Regular Expression 95
Regular Grammar 85
Rehashing 122
REPROC 25

Scanner 2, 12, 35, 105, 153
Scanner, listing of 178
SCORE 18
Screen Editor 156
Sentence 80
Sentences 79
SETRAN 25
SETVAR 25
SKIP 19
SLEEP 25
SMALLER 26
Source Program 12
Statement 21
Statement, conditional 23
Strings 5, 10, 32, 32
SUB 25
Subgraphs 111
Syntax 20, 33, 123, 6

TESTCL 26
Text Compression 114, 10
Transition 5
Transition Diagram 92
Transition Graph 5, 8
Transition Table 31, 31
Transitions, conditional 16
Transitions, random 16
Transitive Verbs 5, 15
Trees 112
Type 0 Grammar 86

Variables 18
Verbs 5, 7, 15, 33
Verbs, intransitive 15
Verbs, transitive 15
Vertices 106

Warshall, S. 108
Woods, D 1
Words 79

ADL – the birth of a new language

Here is a totally new approach to writing adventure games. It does away completely with BASIC, enabling even a novice to write successful adventures.

No conventional programming experience is required and no knowledge of hardware or software. Instead, using Adventure Description Language, you write an adventure as text files containing:

- a map of the locations in your adventure.
- the strings that describe the locations.
- a dictionary defining the words available to the player.
- a description of these words.

The text is translated into compact and efficient adventure code, using a special program called the *scanner.* The code is then executed using *the interpreter.* Full listings of these are included, complete with a *screen text editor.*

All the programs in this book are suitable for a BBC Computer equipped with one or more disk drives. However, the design of the scanner and interpreter are described in great detail from both theoretical and practical standpoints, enabling users of other computers to design their own ADL.

A major strength of this book is that it gives a thorough introduction to the process of designing a computer language. As such, it will be of interest to students of computer science, as well as to enthusiasts of adventure games.

About Sigma

We're still expanding! Book proposals and requests for catalogues should be sent to:

Sigma Press
5 Alton Road
Wilmslow
Cheshire
SK9 5DY

GB £ NET +008.95

ISBN 1-85058-011-1

00895

9 781850 580119